



Synaptics RMI4 Specification

PN: 511-000136-01 Rev. E

Copyright

Copyright © 2007 – 2011 Synaptics Incorporated. All Rights Reserved.

Trademarks

Synaptics, the Synaptics logo, ClearPad and TouchPad are trademarks of Synaptics Incorporated.


All other brand names or trademarks are the property of their respective owners.

Notice

This document contains information that is proprietary to Synaptics Incorporated. The holder of this document shall treat all information contained herein as confidential, shall use the information only for its intended purpose, and shall protect the information in whole or part from duplication, disclosure to any other party, or dissemination in any media without the written permission of Synaptics Incorporated.

Conventions used in this document

The table below describes the documentation conventions. These conventions are used in all Synaptics technical literature.

<i>Term</i>	<i>Meaning</i>
\$	Hexadecimal numbers are marked with a leading '\$' sign: The number \$7FF is equal to 2047 decimal.
—	Bits shown as “—” in register diagrams are equivalent to bits marked <i>Reserved</i> .
<i>italics</i>	<i>Italicized</i> words introduce a term described in the adjacent text or in the Glossary.
<i>Reserved</i>	<i>Reserved</i> is used to signify a bit or bit-field not currently used in any (published) way.
Courier	Courier font is used for text to be entered on a command line or in a program, or for text output from a device.
	This “caution” icon is used to indicate information about something bad that might happen if the guidelines for usage are not followed, or if care is not taken.

Contents

1.	INTRODUCTION	8
1.1.	Conventions used in RMI documentation	8
2.	THE STRUCTURE OF RMI	9
2.1.	RMI functions	9
2.1.1.	Function numbers	10
2.2.	Registers	10
2.2.1.	Query registers	11
2.2.2.	Control registers	11
2.2.3.	Data registers	12
2.2.4.	Command registers	12
2.3.	Register map organization	14
2.3.1.	Register address pages	14
2.3.2.	Global registers	14
2.3.3.	Register grouping within an RMI function	15
2.3.4.	Function grouping within a page	15
2.3.5.	Page Description table	15
2.3.6.	Replicated registers	18
2.4.	Register map organization summary	19
2.5.	RMI physical layer operations	21
2.5.1.	Writing registers	21
2.5.2.	Reading registers	21
2.5.3.	Signaling attention and interrupts	21
2.5.4.	Robust operation	22
2.6.	Register coherence	22
2.6.1.	Write access into coherent regions	23
2.6.2.	Read access into coherent regions	23
2.7.	Data reporting	23
2.7.1.	Interrupt requests	23
2.7.2.	Attention signal	24
2.7.3.	Spontaneous resets	25
3.	FUNCTION \$01: RMI DEVICE CONTROL	26
3.1.	Function \$01: query registers	26
3.1.1.	F01_RMI_Query0: Manufacturer ID query	26
3.1.2.	F01_RMI_Query1: Product Properties query	26
3.1.3.	F01_RMI_Query2, 3: Product Info query	27
3.1.4.	F01_RMI_Query4 through 10: Device Serialization queries	27
3.1.5.	F01_RMI_Query11 through 20: Product ID queries	28
3.1.6.	F01_RMI_Query21: reserved for Synaptics use	28
3.1.7.	F01_RMI_Query22: Sensor ID	28
3.2.	Function \$01: control registers	29
3.2.1.	F01_RMI_Ctrl0: Device Control register	29
3.2.2.	F01_RMI_Ctrl1.*: Interrupt Enable register	31
3.3.	Function \$01: data registers	32
3.3.1.	F01_RMI_Data0: Device Status register	32
3.3.2.	F01_RMI_Data1.*: Interrupt Status register	33
3.3.3.	Function \$01: interrupt source	34
3.4.	Function \$01: command registers	35
3.4.1.	F01_RMI_Cmd0: Device Command register	35

4.	FUNCTION \$08: BIST	36
4.1.	Typical BIST usage scenario	36
4.2.	Function \$08: query registers.....	37
4.3.	Function \$08: control registers	38
4.4.	Function \$08: data registers.....	39
4.4.1.	Function \$08: interrupt source	40
4.5.	Function \$08: command registers	41
5.	FUNCTION \$09: BIST	42
5.1.	Typical BIST usage scenario	42
5.2.	Function \$09: query registers.....	43
5.3.	Function \$09: control registers	44
5.4.	Function \$09: data registers.....	45
5.4.1.	Function \$09: interrupt source	46
5.5.	Function \$09: command registers	47
6.	FUNCTION \$11: 2-D SENSORS	48
6.1.	Number of 2-D sensors	48
6.2.	Function \$11: query registers.....	49
6.2.1.	F11_2D_Query0: per-device query registers	49
6.2.2.	F11_2D_Query1 through 10: per-sensor query registers.....	50
6.3.	Function \$11: control registers	55
6.3.1.	F11_2D_Ctrl0: general control.....	55
6.3.2.	F11_2D_Ctrl1: palm and finger control	57
6.3.3.	F11_2D_Ctrl2, 3: distance threshold	57
6.3.4.	F11_2D_Ctrl4: velocity control.....	58
6.3.5.	F11_2D_Ctrl5: acceleration control	58
6.3.6.	F11_2D_Ctrl6 through 9: maximum X and Y position control.....	58
6.3.7.	F11_2D_Ctrl10, 11: gesture control.....	58
6.3.8.	F11_2D_Ctrl12.*: sensor mapping control	59
6.3.9.	F11_2D_Ctrl13.*: reserved for Synaptics use	60
6.3.10.	F11_2D_Ctrl14: sensitivity adjustment	61
6.3.11.	F11_2D_Ctrl15: maximum tap time	61
6.3.12.	F11_2D_Ctrl16: minimum press time	61
6.3.13.	F11_2D_Ctrl17: maximum tap distance.....	61
6.3.14.	F11_2D_Ctrl18: minimum flick distance	62
6.3.15.	F11_2D_Ctrl19: minimum flick speed	62
6.4.	Function \$11: data registers.....	63
6.4.1.	Data register layout	63
6.4.2.	Finger reporting	64
6.4.3.	F11_2D_Data0.*: finger status data.....	65
6.4.4.	F11_2D_Data1, 2: X and Y position data (MSB).....	65
6.4.5.	F11_2D_Data3: X and Y position data (LSB)	65
6.4.6.	F11_2D_Data4: finger width (W) data	66
6.4.7.	F11_2D_Data5: finger contact (Z) data	66
6.4.8.	F11_2D_Data6.*, 7.*: finger motion deltas.....	66
6.4.9.	F11_2D_Data8, 9: gesture data	67
6.4.10.	F11_2D_Data10: pinch motion and X flick distance.....	69
6.4.11.	F11_2D_Data11: rotate motion and Y flick distance	69
6.4.12.	F11_2D_Data12: finger separation and flick time	70
6.4.13.	F11_2D_Data13.*: TouchShape status	70
6.4.14.	F11_2D_Data14: x lower scroll motion / MultiFinger horizontal scroll	70

6.4.15.	F11_2D_Data15: y right scroll motion / MultiFinger vertical scroll	70
6.4.16.	F11_2D_Data16: x upper scroll motion	71
6.4.17.	F11_2D_Data17: y left scroll motion	71
6.4.18.	Function \$11: interrupt source	71
6.5.	Function \$11: command registers	72
7.	FUNCTION \$19: 0-D CAPACITIVE BUTTONS	73
7.1.	Function \$19: query registers	73
7.1.1.	F19_Btn_Query0: Configurable button query	73
7.1.2.	F19_Btn_Query1: ButtonCount query	73
7.2.	Function \$19: control registers	74
7.2.1.	Calculating the number of control registers	74
7.2.2.	F19_Btn_Ctrl0: general control	75
7.2.3.	F19_Btn_Ctrl1.*: button interrupt enable control	76
7.2.4.	F19_Btn_Ctrl2.*: single button participation control	76
7.2.5.	F19_Btn_Ctrl3.*: sensor map control	76
7.2.6.	F19_Btn_Ctrl4.*: reserved	78
7.2.7.	F19_Btn_Ctrl5: all-button sensitivity adjustment	78
7.2.8.	F19_Btn_Ctrl6: all-button hysteresis threshold	78
7.3.	Function \$19: data registers	79
7.3.1.	Calculating the number of data registers	79
7.3.2.	Function \$19: interrupt source	79
7.4.	Function \$19: command registers	80
8.	FUNCTION \$30: LED/GPIO CONTROL	81
8.1.	Function \$30: power management	81
8.2.	Function \$30: query registers	82
8.3.	Function \$30: control registers	83
8.3.1.	Calculating the number of control registers	83
8.3.2.	F30_GPIO_Ctrl0.*: GPIO/LED select	83
8.3.3.	F30_GPIO_Ctrl1: GPIO/LED general control	84
8.3.4.	F30_GPIO_Ctrl2.* and F30_GPIO_Ctrl3.*: GPIO input/output mode control	84
8.3.5.	F30_GPIO_Ctrl4.*: LED active control	85
8.3.6.	F30_GPIO_Ctrl5.*: LED ramp period control	86
8.3.7.	F30_GPIO_Ctrl6.*: GPIO/LED control	86
8.3.8.	F30_GPIO_Ctrl7.*: button-to-GPIO mapping and control	89
8.3.9.	F30_GPIO_Ctrl8.*: haptic enable control	90
8.3.10.	F30_GPIO_Ctrl9: haptic duration control	90
8.4.	Function \$30: data registers	91
8.4.1.	Function \$30: interrupt source	91
8.5.	Function \$30: command registers	92
8.6.	Function \$30 example: complete control layout	92
8.7.	Function \$30 examples: query bits	93
8.7.1.	Function \$30 example: both GPIOs and LEDs	93
8.7.2.	Function \$30 example: LEDs only	93
8.7.3.	Function \$30 example: GPIOs only, with driver control	93
8.7.4.	Function \$30 example: GPIOs only, without driver control	93
9.	FUNCTION \$32: TIMER	94
9.1.	Function \$32: query registers	94
9.1.1.	F32_Timer_Query0: Timer properties query	94
9.2.	Function \$32: control registers	95

9.2.1.	F32_Timer_Ctrl0, 1: timer match count, mode, and enable	95
9.3.	Function \$32: data registers.....	97
9.3.1.	F32_Timer_Data0, 1: Timer count, match, and run	97
9.3.2.	Function \$32: interrupt source	98
9.4.	Function \$32: command registers	98
10.	FUNCTION \$34: FLASH MEMORY MANAGEMENT	99
10.1.	Overview.....	99
10.1.1.	Non-Volatile Memory Organization	99
10.1.2.	Modality.....	100
10.2.	Function \$34: query registers.....	102
10.2.1.	F34_Flash_Query0, 1: Bootloader ID query	102
10.2.2.	F34_Flash_Query2: Flash Properties query	102
10.2.3.	F34_Flash_Query3, 4: Block Size query.....	102
10.2.4.	F34_Flash_Query5, 6: Firmware Block Count query	103
10.2.5.	F34_Flash_Query7, 8: Configuration Block Count query	103
10.3.	Function \$34: data registers.....	104
10.3.1.	F34_Flash_Data0,1: Block Number registers	104
10.3.2.	F34_Flash_Data2.*: Block Data registers.....	104
10.3.3.	F34_Flash_Data3: Flash control/status register.....	104
10.3.4.	Function \$34: interrupt source	108
10.4.	Flash programming procedures	108
10.4.1.	Bootloader mode and ATTN.....	108
10.4.2.	Enable flash programming.....	108
10.4.3.	Program the firmware image	109
10.4.4.	Program the configuration image.....	110
10.4.5.	Disable Flash Programming mode.....	110
10.5.	Configuration space layout.....	111
11.	FUNCTION \$36: AUXILIARY ADC	113
11.1.	Function \$36: query register	113
11.2.	Function \$36: control register.....	114
11.3.	Function \$36: data registers.....	115
11.4.	Function \$36: command registers	116
11.5.	Function \$36: interrupt source.....	117
12.	STANDARD RMI PHYSICAL LAYERS.....	118
12.1.	I ² C physical interface	118
12.1.1.	I ² C transfer protocols.....	118
12.1.2.	RMI register addressing	119
12.1.3.	Block read operations.....	119
12.1.4.	Block write operations	119
12.1.5.	I ² C protocol compliance.....	120
12.2.	SMBus physical interface.....	121
12.2.1.	RMI-on-SMBus addressing	121
12.2.2.	SMBus transfer protocols	121
12.2.3.	Multi-register block read/write operations	123
12.2.4.	Repeated starts	123
12.2.5.	SMBus compliance.....	123
12.3.	SPI physical interface	125
12.3.1.	SPI signals	125
12.3.2.	SPI clocking	125
12.3.3.	SPI transaction format.....	126

12.3.4. SPI attention mechanism.....	127
--------------------------------------	-----

1. Introduction

This document defines a register-oriented protocol for use in Synaptics® embedded products. The overall protocol is known as RMI: the Register Mapped Interface. RMI uses a “register map” model that is convenient and familiar to host system developers.

The basic goals of RMI are:

1. To support a large and varied product line, with an emphasis on forward-, backward-, and cross-compatibility and consistency among Synaptics products
2. To employ industry-standard I²C, SMBus, and SPI-based interfaces, following the familiar and easy-to-use “register” model that is commonly found in devices with these interfaces.
3. To be easy to document, understand, and use from the perspective of implementers of RMI drivers and systems incorporating RMI devices. RMI is designed so that any given RMI product can be documented concisely. For example, the numbering of functions and data sources is elaborate when considering the RMI protocol as a whole, but in each specific RMI device the resulting register map is straightforward and easy to use.

Each RMI product uses a particular physical interface (I²C, SMBus, or SPI) to access a particular register set tailored to the product. But RMI itself is a platform protocol that ties together the common aspects of all physical interfaces and all register maps of Synaptics’ various embedded products.

1.1. Conventions used in RMI documentation

Bits within a byte, register, or other quantity are numbered with bit 0 as the least significant bit of the register or quantity. Ranges of bits are denoted $n:m$ for the field of bits numbered n down to m , inclusive. For example, bits 7:4 comprise the most significant four bits of an 8-bit register.

All signed quantities in RMI are expressed in two’s complement hexadecimal notation, where the most significant bit is taken as a sign bit. For example, a signed 8-bit byte is \$00 to encode 0, \$7F to encode +127, \$80 to encode –128, and \$FF to encode –1. A signed 6-bit register field would be \$00 to encode 0, \$1F to encode +31 (the largest value that can be encoded in a signed 6-bit field), \$20 to encode –32 (the smallest value that can be encoded), and \$3F to encode –1.

2. The structure of RMI

RMI, the Register Mapped Interface, is a communications interface for use with Synaptics modules. RMI is built upon industry-standard physical interfaces. Initially, RMI offers a choice of I²C, SMBus, or SPI. RMI communications involve two entities: The *host*, typically the main system processor, is the master. The *device*, typically a chip or module supplied by Synaptics, is a slave.

For systems with multiple hosts or multiple devices, RMI relies on the arbitration and addressing mechanisms of the underlying physical interface. For example:

- In SPI-based RMI systems with multiple devices, the host might generate a separate SSB signal for each device.
- In I²C or SMBus-based RMI systems with multiple hosts, the hosts might use bus arbitration and Repeated-Start transactions to negotiate safe shared access to a device.

2.1. RMI functions

RMI defines a standard set of *functions* that define features such as 2-D TouchPad™ sensors and brightness-controlled LEDs. The features and capabilities of an RMI device arise from the set of RMI functions that are included in the device. A particular RMI-based product will include or omit each possible function depending on the specific needs of the product. Certain RMI functions may only be supported on specific Synaptics ASICs. For example, an RMI function to allow in-system reprogramming of Flash will only be available on Flash-based ASICs. Each function is largely independent of any other functions that might be present in the same device.

Functions are consistently defined among all devices that include them. For example, Function \$01 always corresponds to general device control features, and the registers associated with Function \$01 are defined in a consistent way in all RMI devices that include Function \$01. For devices that do not require Function \$01, the registers associated with Function \$01 are unimplemented.

Some RMI functions are completely universal, with a fixed definition that is identical in any product that includes them. Other RMI functions represent more general capabilities, with parameters that may be chosen differently from one product to another. For example, a function that supports 1-D Sensor operation may involve one linear strip sensor in one product, and two closed-loop sensors in another. Still other RMI functions may define flexible capabilities, such as the ability to specify the functionality of a device via run-time configuration methods. In general, each RMI function defines the set of:

- The control registers that it needs to configure its operation modes,
- The data registers that it uses to report information back to the host,
- The interrupt sources that it uses to signal changes in the contents of its data registers,
- The command registers that it might need to implement function-specific commands, and
- The query registers that would enable a host to learn about its specific capabilities.

RMI functions are not required to implement all of these registers. The specific documentation for each RMI function defines the set of registers implemented by that function, as well as their contents.

2.1.1. Function numbers

RMI functions are identified by an informal name and a standardized identifying number. Function numbers are used to identify the capabilities supported by a device. The identifying number for an RMI function is an 8-bit integer in the range \$01–\$FF. The standard function numbers are shown in Table 1.

Table 1. Standard function numbers

Function	Purpose	See page
\$01	RMI Device Control	26
\$08	BIST	36
\$09	BIST	42
\$11	2-D TouchPad sensors	48
\$19	0-D capacitive button sensors	73
\$30	GPIO/LEDs (includes mechanical buttons)	81
\$32	Timer	94
\$34	Flash Memory Management	99

2.2. Registers

RMI is defined in terms of a set of logical *registers* that appear within a register address map. The host communicates with the device by reading and writing the device’s registers through physical interface transactions.

All registers in RMI are 8 bits wide. Quantities larger than a byte are held in several consecutive registers which are typically read or written as a group.

Certain multi-byte quantities may *require* that the host must read or write them as a group, called a *coherent region*. The general rules regarding the read and write access of a coherent region are discussed in section 2.6.

Registers are identified by 16-bit addresses. Where ‘\$’ signifies hexadecimal notation, the register addresses range from \$0000 to \$FFFF. Each address in this range potentially identifies one byte of register data. Only a few of the 65536 potential addresses are actually implemented; other addresses are marked *reserved*.

The register address space is divided into *pages* of related registers. Each page consists of 256 registers in the range \$xx00–\$xxFF.

RMI devices typically export their entire customer-oriented register set within the first page of register addresses, covering the address range \$0000–\$00FF. See section 2.3 for details on the register address map organization.

Although in principle an RMI device can do anything in response to a read or a write of any register, RMI defines several standard types of registers with well-specified behaviors:

- Query registers,
- Control registers,
- Data registers, and
- Command registers.

These are described in the following sub-sections.

2.2.1. Query registers

Query registers are read-only registers that allow the host driver to determine what kind of RMI device is attached and what features it includes. Most hosts in embedded systems will never need to read the query registers at all, but platform host drivers and diagnostic tools may find these registers useful.

Query registers typically report constant data read from ROM on the device. Query registers may also report information that can change dynamically, based on how a device might have been configured.

The host should not write to a query register, but in any case writes to query registers are ignored.

Reserved query bits typically read as ‘0’, but the host should ignore reserved query bits for forward compatibility with future RMI devices that might use these bits for additional query information.

2.2.2. Control registers

Control registers allow the host to initialize the device and control its functions. A control register generally looks like a readable and writable RAM location. The host can write data to the register, and the host can read the current contents of the register back without side effects.

Control registers generally do not change except when explicitly written by the host. However, this is merely a guideline and not a strict requirement: An RMI product or function may define control-like registers that change for other reasons. An example of this would be that a control register also reports status information, where the status information might change spontaneously.

Each control register has a defined reset value. When the device resets for any reason, all the control registers revert to their reset values. Flash-based products may allow for in-system changes to these default reset values. Consult the product-specific documentation to see which control registers allow flash-settable defaults.

Writing to a control register generally affects some aspect of the device’s operation, either immediately or at a later time. Some control registers may also have side effects upon writing. Any such side effects are described in the documentation for the register.

A control register typically holds some *parameter* that configures the device. Parameters may have different sizes. The smallest parameter would be represented as a single bit. Larger parameters could fill an entire register.

Parameters larger than the size of a single register are allowed to span multiple registers. Sometimes the fixed properties of a particular device, such as the number of strip sensors in a 1-D function, are also referred to in RMI as *parameters*. Some parts of a control register may be marked *reserved* or “—”, and some whole registers in a group of control registers may be *reserved*. Reserved bits normally reset to ‘0’; these bits may harmlessly be written to ‘0’, but the behavior of the device is undefined if the host writes a reserved bit to ‘1’. For example, some reserved control bits may activate undocumented or proprietary features of the device when written to ‘1’, and those undocumented features may change without notice from one version of the product to another.

Similarly, some possible settings of a control register may be marked *reserved*, and the behavior of the device is undefined if the host sets a register to a reserved value.

Some control registers or parts of control registers are implemented only in some versions or models of a device. When a control register is *unimplemented*, it resets to a suitable value reflecting the fixed behavior that is implemented in its place (for example, a fixed sensitivity setting, or a fixed ‘0’ enable bit for an unimplemented mode).

At the implementation's discretion, an unimplemented register or register bit may be treated as a query register (that is, writes are ignored regardless of the data written), or as a control register that does nothing (that is, writes change the contents of the register but have no other effect on the device). In all cases, writes to unimplemented control bits are harmless.

Similarly, some possible settings of a control register may be unimplemented in some versions or models of a device; writing a control register to a setting that is unimplemented on the device has an undefined effect, but the effect is generally harmless and most often corresponds to one of the implemented settings of the register.

2.2.2.1. Device configuration

The complete set of parameters (both fixed and adjustable) contained in the control registers of an RMI device is together called the *configuration* of the device. A device can be *configured* at run-time by writing each control register in the device with the appropriate configuration data.

The amount of run-time configuration required will be specific to both the product and the application it is being used for. Some RMI products used in certain applications may not require any run-time configuration.

Flash-based products may contain their default device configuration in a special area of the Flash that can be updated without requiring a full firmware update. This would allow the configuration to be updated in-system. For more information, see section 10 and consult the product-specific documentation.

2.2.3. Data registers

Data registers report sensor readings and other input data to the host. Data registers are typically read-only in nature. Data registers may also support special semantics such 'read/clear' where a certain bit or bits within the data register might be automatically cleared after the register is read. Data registers can generate interrupt requests at certain times or at a certain rate. The special structure and behavior of RMI data registers are detailed in section 2.7.

Data registers may be defined to contain status information that might change spontaneously during device operation. Note that spontaneous changes to the status fields in a data register are capable of generating interrupts, while spontaneous changes to the status fields in a control register are not.

Some parts of some data registers are marked *reserved*; host software should always ignore these bits. RMI devices typically report '0' in all *reserved* data register bits, but the host should not rely on this.

2.2.4. Command registers

Command registers are read/write registers that allow the host to perform discrete commands or to signal discrete events on the device. Each bit in a command register corresponds to a possible command. Command register bits are special in that the host can only write them from '0' to '1'. Writing a '0' to a command bit leaves the state of the bit untouched by the write operation. Writing a '1' to a command bit issues the command. The command bit automatically clears to '0' when the command completes.

Some commands may complete instantaneously; their bits will never read as '1'. Other commands may take some time to complete. A host may either poll the command register to see when the command is complete or it may proceed immediately knowing that the command will execute in due course.

Certain command register bits may also be set to '1' by the device itself. RMI does not define what happens if a '1' is written to a command bit that is still '1' from a previous posting of the same command. The behavior depends on the particular command and function. For this reason, the host should never use a read/modify/write operation to write a bit or bits in a command register.

It is acceptable to write a '1' to one command bit when other command bits are already '1' due to previously posted, still-pending commands. The result is that several commands will be posted at once. The order in which the device executes these pending commands is implementation-defined, and may not be the same as the order in which the commands were posted. In situations where the order of command execution is significant, the host should read and wait until the command register becomes \$00 before writing a new command.

A command bit that causes an RMI device to reset will be irregular in that the RMI host interface will go "off the air" when the reset command is posted. After posting a reset command, it is not meaningful to poll the command register to wait for completion of the command, nor to post another command at the same time as a reset command is pending.

Unused bits in a command register are marked *reserved* or *unimplemented*. The host must never write a '1' to a reserved or unimplemented command bit. For example, some reserved command bits may activate undocumented or proprietary features of the device when written to '1' and these undocumented or proprietary features are subject to change without notice. Reserved or unimplemented command register bits, and wholly reserved or unimplemented command registers, are not required to behave as described in this section when written to '1'. Writing \$00 to a command register has no effect, and is harmless.

2.3. Register map organization

An RMI device always contains one or more RMI functions. Each of the RMI functions present in a device define the set of control, command, data, and query registers required to implement their own specific functionality. The comprehensive register map for a given device is formed by organizing the registers defined by each function that it implements into a single, orderly register map. The organization process may be product dependent, but will always be deterministic in nature.

2.3.1. Register address pages

All registers defined by a particular RMI device are placed within a general 16-bit RMI register address space. This 16-bit RMI register map is divided into pages of 256 registers per page. The register page address defines the upper eight bits of the 16-bit RMI address. The register page offset, or more simply, the offset defines the lower eight bits of the 16-bit RMI address. A single page can contain the registers associated with one or more RMI functions. Register pages are defined to be self-contained:

- All registers defined by a particular RMI function must live on a single page.
- RMI forbids reading or writing a sequence of registers that goes past the last address in a page. The effects of attempting to read or write beyond the end of a page are undefined.

By convention, RMI products will make every effort to place all of the customer-facing RMI functions on page \$00. From a customer's point of view, this means that a typical RMI device will appear to have an 8-bit address space. Proprietary RMI functions like diagnostic or manufacturing mode functions will typically not live in page \$00.

2.3.2. Global registers

Global registers are defined to appear at the same offset within every page. These registers perform the same basic function regardless of which page they exist in. The Page Select register and the query registers collectively known as the Page Description table are global registers.

2.3.2.1. Page Select register

RMI registers are always addressed with a unique 16-bit address. However, not all physical layers are capable of supplying the full 16 bits of register address information in a single transfer. RMI defines that at a minimum, all physical layer implementations must be capable of supplying the low-order 8 bits of the full 16-bit RMI address. The Page Select register is used to supply any high-order address bits that the physical layer is incapable of sending. For example, SMBus physical layer transfers are defined to supply the low-order 8 bits of address information as part of each transfer. For SMBus transfers, the Page Select register supplies the high-order 8 address bits of the register address. In contrast, SPI physical layer transfers supply the low order 15 bits of the 16 RMI address bits, so the Page Select register need only supply the most significant bit of the 16-bit RMI address. For SPI, bits 6 through 0 of the Page Select register are ignored.

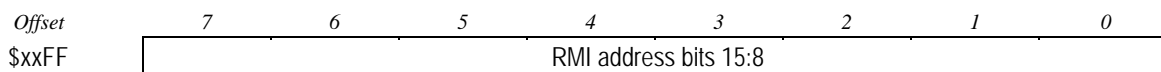


Figure 1. Page Select register

The Page Select register is defined to exist at the same page offset on every RMI address page. This means that the Page Select register can always be accessed, regardless of its current contents. The default page offset for the Page Select register is product-specific, although typically it is at offset \$FF.

The default value of the Page Select register typically is \$00.

2.3.3. Register grouping within an RMI function

The organization of a device's register map occurs at a few different levels. At the lowest level, RMI functions are always designed so that the various kinds of registers (control, data, command, and query) associated with a function are always grouped together in like-kind groups.

For example, all the data registers defined by a particular RMI function are defined as a single group of data registers with sequentially increasing address offsets.

The ordering of the registers within each like-kind group is strictly defined by the function's specification. This means that by knowing the start offset for a particular kind of register group associated with a particular function, the offsets of all of the other like-kind registers associated with that function can be determined.

2.3.4. Function grouping within a page

If a page contains more than one RMI function, then the like-kind groups of registers defined by each function are typically grouped together to form like-kind metagroups. For example, all groups of data registers defined by the functions on a page are grouped together in a single data metagroup. This approach allows a host to read the entire set of data registers located on a page in a single physical layer transfer, or to write a complete configuration to all control registers on a page in a single physical layer transfer.

The ordering of the groups within a metagroup is unspecified. Typically, the ordering of the metagroups is the data register group first (lowest page offsets), followed by the control register group, then the command register group, followed by the query register group. Usually there is no empty space between the metablocks.

2.3.5. Page Description table

The mechanisms describing the organization of registers within a function, and the organization of functions within a page, are sufficient to implement RMI products. However, the result of the organization processes is necessarily product-specific. While a typical customer will choose to write a product-specific driver for the particular product, there may also be situations where a single host may be required to interface to a variety of RMI devices.

RMI defines a discovery mechanism that allows an interested host to discover what RMI functionality is present in any particular RMI device. Not only that, the discovery mechanism supplies enough information to enable a host to reconstruct a complete view of the RMI register address map for any given RMI device.

The discovery mechanism is page-based. For devices that support the discovery mechanism, each page with at least one RMI function contains a set of special read-only registers within a Page Description table.

If a host reads the contents of the Page Description table on a particular address page, it can discover:

- what RMI functions exist on that page
- the starting page offsets for each kind of register block associated with each RMI function on that page
- the number of interrupt sources associated with each RMI function on that page
- the version of each RMI function on that page

By scanning for Page Description tables in the general RMI register map, a host can reconstruct the entire RMI register address map, along with the interrupt bit assignments in the RMI Interrupt Status and Interrupt Enable registers. RMI devices that are severely ROM constrained are permitted to omit the Page Description table.

The Page Description table is defined as a general properties query register, followed by an array of Function Descriptors. The contents of the Page Description table are stored from high page offsets down towards lower page offsets, starting at page offset PdtTop.

The PdtTop typically has a value of \$EF. Products that place the top of their Page Description table somewhere other than address offset \$EF will define the new location in their product-specific documentation. A pictorial view of the relationship between the Page Description table and the RMI register set in a device can be found in section 2.4.

Because the value \$00 does not belong to the range of permissible RMI function numbers, the end of the Page Description table is marked by finding a Function Descriptor containing an RMI function number of \$00.

2.3.5.1. Query register 0: Page Description table properties

This register describes some basic information regarding the Page Description table itself.

Offset	7	6	5	4	3	2	1	0
PdtTop-0	—	NonStdPSR	—	—	—	—	—	—

Figure 2. Page Description table Properties Query register

The bits of this register are defined as follows:

NonStdPSR (QueryBase+0, bit 6)

When '1', this bit indicates that the Page Select register is located somewhere other than its standard location at page offset \$FF. Under those circumstances, the location of the Page Select register is defined in the product-specific documentation.

The rest of the bits in this register are *reserved* for future use.

2.3.5.2. Query registers 1-N: Function Descriptors

The Function Descriptor query registers live in the Page Description table, starting at the address offset PdtTop-1. These registers are defined in such a fashion that a host can discover exactly what RMI functions live on that page, where to find the various kinds of registers defined by each of those functions, and the interrupt source count for those functions. This is enough information to reconstruct the complete RMI-compliant address map for any RMI device.

Function Descriptors are defined as follows:

Offset	7	6	5	4	3	2	1	0
FuncDescriptor-0	Function Number							
FuncDescriptor-1	—	Function Version		—	—	Interrupt Source Count		
FuncDescriptor-2	DataBase							
FuncDescriptor-3	ControlBase							
FuncDescriptor-4	CommandBase							
FuncDescriptor-5	QueryBase							

Figure 3. Function Descriptor registers

The first register in the Function Descriptor defines the RMI function number that is implemented within the current page of the address space. A function number with the value \$00 indicates the end of the function descriptor array.

The second register in the Function Descriptor contains some miscellaneous information about the function:

- The *Function Version* field is typically '0'. A non-zero value indicates that the definition of the function has changed in some fashion that is not backwards compatible with the function's original specification. An updated specification will define the changes between function versions.
- The *Interrupt Source Count* defines the number of interrupt source bits that this RMI function needs to allocate within the RMI Interrupt Enable and Interrupt Status registers. The value 7 is reserved to indicate "more than 6 interrupt sources."

Interrupt source bits are allocated by scanning all pages in a device that contain Page Description tables (PDTs), starting at page \$00xx. As each PDT is encountered, the interrupt source bits are allocated in the order that the Function Descriptors are encountered within the Function Descriptor table, starting from the Function Descriptor placed at the highest address in the table. In particular: the first Function Descriptor encountered that defines a non-zero count for its number of interrupt sources is assigned bit positions starting with bit 0 in the Interrupt Status and the Interrupt Enable registers. Subsequent descriptors that allocate additional interrupt sources are allocated at the first available least-significant interrupt source bit.

For example: An RMI device contains three hypothetical functions, Function \$10, Function \$20, and Function \$30, appearing in the function descriptor table in that sequence. Assume that Function \$10 defines two interrupt sources.

Function \$20 defines zero interrupt sources, and Function \$30 defines three interrupt sources. Function \$10 gets processed first, so it allocates the first available least-significant interrupt bits: bits 0 and 1. Function \$20 gets processed next, but does not allocate any interrupt request bits. Function \$30 gets processed last. Since interrupt bits 0 and 1 have already been allocated, Function \$30 allocates the least-significant interrupt bits that are still available: bits 2, 3, and 4.

This means that the Interrupt Status register and Interrupt Enable registers for that product appear as:

Offset	7	6	5	4	3	2	1	0
Interrupt Enable Reg	---	---	---	F30 IEN2	F30 IEN1	F30 IEN0	F10 IEN1	F10 IEN0

Figure 4. A sample Device Control register, showing bit assignments

If there are more interrupt bits allocated than will fit into a single register, then additional registers are allocated to hold them. This means that a complex RMI device may contain multiple Interrupt Enable and Interrupt Status registers. Devices that need to define multiple interrupt registers always define the additional registers at subsequent locations in the register map. Multiple registers of the same type are also known as *replicated registers*, and are described in section 2.3.6.

The third through sixth registers in the Function Descriptor define the base address offsets on the current page for each of the four kinds of registers groups. The PDT only supplies the base address for the different kinds of registers. The interpretation of the register space after the base address must be done in accordance with the definition of each function as described in the RMI specification. In particular, the contents of an RMI function query register may alter the interpretation of the register space.

In terms of base addresses, PDT entries should be considered to be totally separate from each other. For example, an RMI device may implement two functions that support one command register each.

The PDT entry for Function 1 might start the command register at offset \$10, while the subsequent PDT for Function 2 is allowed to place its command register at offset \$20. But a driver scanning this PDT should not make any inferences regarding Function 1, such as assuming that it implements \$20 - \$10 or the \$10 command registers.

If the RMI specification for a particular function does not actually define a particular kind of register, the base address contents for that register kind is meaningless and should be ignored.

2.3.6. Replicated registers

RMI allows for the replication of either single registers or blocks of related registers in the register map. The replicated block may vary in size, but the registers or register blocks within are all identical in form. The replication count can always be determined from information contained in one or more query registers. Because of that, a host can properly account for all replicated areas while reconstructing a device register map using the RMI discovery mechanisms.

2.3.6.1. Replicated single registers

The simplest case of replication is where a singular register gets duplicated one or more times in the address map. These simple replicated registers are marked with a “.” in their description in this specification. For example, RMI Function \$11 (2-D) contains a set of replicated Sensor Mapping registers defined as ‘F11_2D_Ctrl12.’. The specification for F11_2D_Ctrl12. states that the number of replicated registers can be determined from the *MaximumElectrodes* field in F11_2D_Query4. Therefore, if a particular product defines *MaximumElectrodes* to be the value 3, three F11_2D_Ctrl12 registers will be defined by the product.

In the register map for that product, those three replicated registers will appear as:

```
F11_2D_Ctrl12.0    Sensor Mapping Control
F11_2D_Ctrl12.1    Sensor Mapping Control
F11_2D_Ctrl12.2    Sensor Mapping Control
```

In the device register map, each of the replicated registers gets a replication suffix appended to the basic register description number, starting at ‘.0’.

The replication formula can be more complex than the simple example above. For example, every RMI product contains at least one F01_RMI_Data1. Interrupt Status register. The actual number of Interrupt Status registers implemented by an RMI device can be calculated by counting the total number of Interrupt Sources in the device:

$$\text{InterruptStatusRegisterCount} = \text{trunc}((\text{NumberOfInterruptSources} + 7) / 8)$$

If the number of interrupt sources in a particular product was 10, the interrupt status register count would be (10+7)/8 or 2. In the register map for that product, those two replicated registers would appear as:

```
F01_RMI_Data1.0    Interrupt Status
F01_RMI_Data1.1    Interrupt Status
```

Also note that just because registers can be replicated, does not mean that they *always* are. There may be instances where the replication count is 1, or even 0. If the interrupt status count in the previous example was only 3, then the register map for that product would show a single Interrupt Status register as F01_RMI_Data1.0.

2.3.6.2. Replicated register blocks

There are times when a related block of registers needs to be replicated. For example, the RMI4 F\$11 function defines a block of five registers (F11_2D_Data1 through F11_2D_Data5). These five registers are used to specify the various pieces of information regarding the absolute position associated with a single finger. If the 2-D sensor is capable of supporting more than one finger (that is, if the *NumberOfFingers* query in F11_2D_Query1 is greater than 0), the block of per-finger reporting registers will be replicated for all subsequent fingers.

For example, if a particular device reports *NumberOfFingers* as '1' (meaning 1+1 or two fingers being reported), then there are two blocks of replicated finger registers:

F11_2D_Data1.0	X Position 11:4
F11_2D_Data2.0	Y Position 11:4
F11_2D_Data3.0	X Posn 3:0, Y Posn 3:0
F11_2D_Data4.0	WX, WY
F11_2D_Data5.0	Z
F11_2D_Data1.1	X Position 11:4
F11_2D_Data2.1	Y Position 11:4
F11_2D_Data3.1	X Posn 3:0, Y Posn 3:0
F11_2D_Data4.1	WX, WY
F11_2D_Data5.1	Z

The replication suffix is appended to each register in the block being replicated, and gets incremented for each subsequent block. In the example above, the '.0' replication suffix indicates the replicated block associated with the first finger, while '.1' indicates the block associated with the second finger.

2.4. Register map organization summary

As mentioned earlier, all RMI devices typically place all customer-centric registers on page \$00. Customers should be able to consider that an RMI device contains a single page of addresses. Proprietary RMI functions that define manufacturing or diagnostic functions are usually placed somewhere other than page \$00.

Each page in the RMI address space is organized as a self-contained unit, organized as follows:

- The Page Select register is placed at a consistent address in each page, as described in 2.3.2.1.
- The registers implemented by the set of RMI functions present on the page are organized according to the rules in 2.3.3 and 2.3.4.
- A Page Description table is [optionally] placed on each page containing at least one RMI function.

Below is a sample register map describing page \$00 of a hypothetical RMI4 device. By scanning the Page Description table downwards starting from register \$EF, the host can determine that there are three RMI functions on the page (F\$11, F\$22, F\$33) and that there were a total of three interrupts defined, so there must be exactly one Interrupt Status register and one Interrupt Mask register.

Note: For the purposes of this example, the functions F\$11, F\$22, and F\$33 and their registers are ‘imaginary’, in that they are not intended to bear any resemblance to a real F\$11, F\$22, or F\$33 that might get defined at some later date. Also in this example, the F\$11 data register 0 implements an Interrupt Status register. The bits within that register would be assigned as:

1. F\$11 is the first function encountered in the Page Description table. Because it defines a single interrupt source, that interrupt source would be assigned bit 0 in the Interrupt Status register.
2. F\$22 is encountered next but defines no interrupt sources, so no bits are allocated for F\$22 in the Interrupt Status register.
3. F\$33 is encountered last. It defines two interrupt sources, so it would be assigned bits 1 and 2 in the Interrupt Status register.

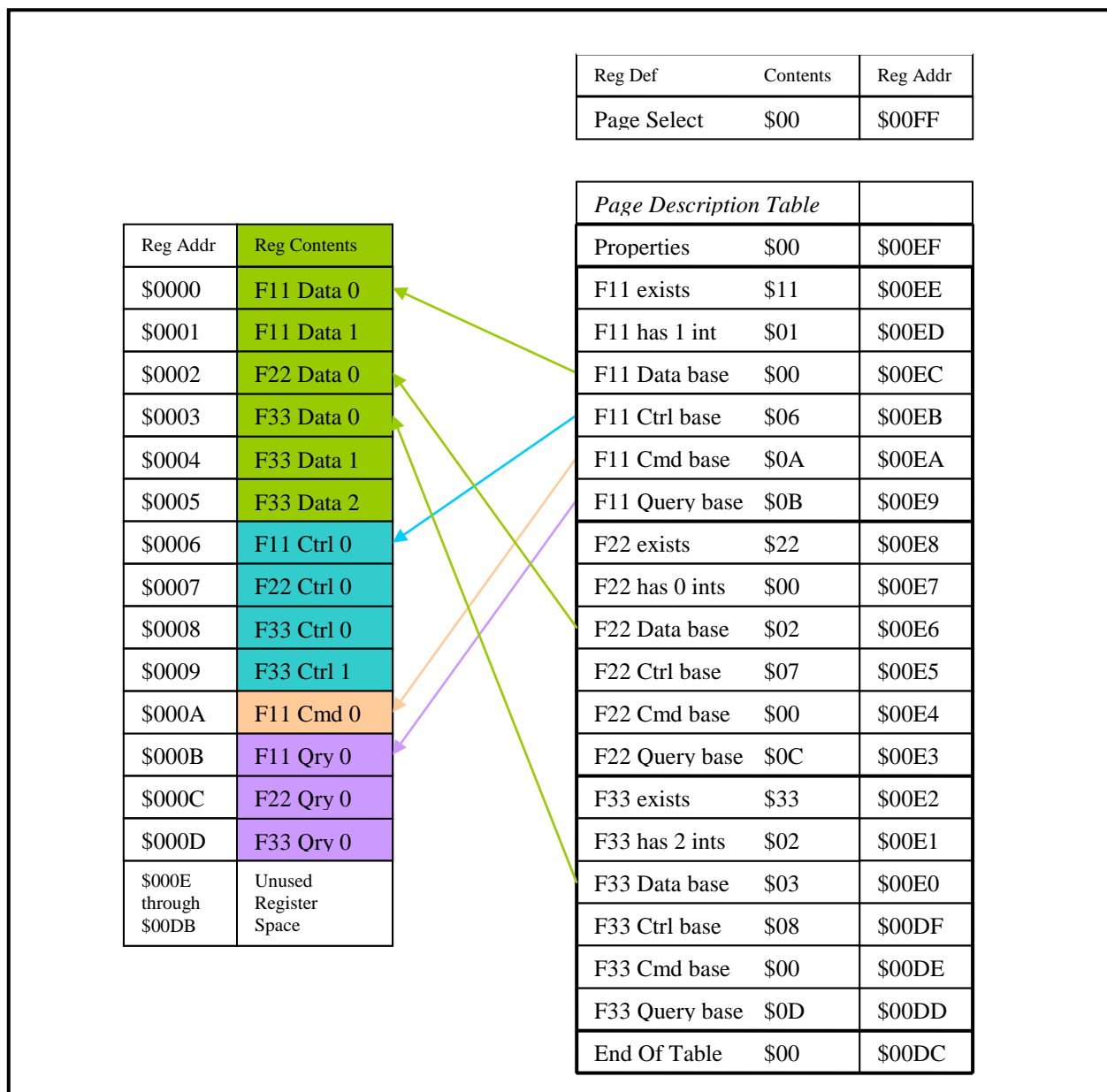


Figure 5

Note: Each Page Description table always contains base pointers for all four kinds of registers. For functions that do not define certain kinds of registers, a base pointer corresponding to an unimplemented kind of register is meaningless, and its value should be ignored. In the example above, if the specification for a hypothetical Function \$22 were to state that Function \$22 defined no command registers, then the value returned by reading the Function \$22 Command Base register should be ignored.



Important: The Page Description table only supplies the base address; to find out the number of registers, register usage, and other information, you must use queries.

2.5. RMI physical layer operations

This section summarizes the basic operations that are supported by every RMI physical layer. For specific details of the physical layers currently defined for RMI, see section 11.

2.5.1. Writing registers

The host may write to any N consecutive device registers starting at any register address R in a single transfer. The write transaction always “succeeds” as far as the RMI protocol is concerned. The value of N can be as small as 1 in order to support certain physical layer protocols like SMBus. The physical layer may also define an upper limit on the value of N . See section 11 for more information.

A single register write operation may not span more than one register page. In other words, the N consecutive registers covered by a given write operation must have addresses that differ only in the low 8 bits. It is an error if the host performs a write operation that spans multiple pages; the effect is undefined and implementation-dependent.

As a special case, a write operation to a command register that causes a device to reset must write only to that command register, and must only set the command bit that initiates the reset operation. Most physical layers define a hold off time after the Reset command before the host can again reliably access the RMI interface.

2.5.2. Reading registers

The host may read from any N consecutive device registers starting at any register address R . The read transaction always “succeeds” as far as the RMI protocol is concerned.

As with the write operations, the high and low limits on the value of N may be dependent on the physical layer implemented by a device. See section 11 for more information. A single register read operation may not span more than one register page (in the sense defined above in section 2.5.1).

Preferably, the physical layer will allow the host to choose to adjust dynamically the number N of registers that it reads based on the first few data bytes it has read. However, RMI never *requires* the host to perform a read operation of non-fixed size, because some hosts lack this ability (for example, because of restrictions in their OS drivers).

2.5.3. Signaling attention and interrupts

The device may signal the host for attention. From the host’s point of view, the attention signal acts like a traditional interrupt signal. The host responds to the attention interrupt by reading the appropriate device registers to determine the reason for the attention request.

2.5.4. Robust operation

Every physical layer should provide for robust system operation in the presence of spontaneous resets at any time by either the device or the host.

Spontaneous resets on the device side are a fact of life in many capacitive touch sensor designs. Touch sensors, by their nature, are more exposed than most electronics to disruption by ESD (electrostatic discharges) during operation, particularly if the sensor area is framed by an open bezel instead of being under solid, uninterrupted plastic. Synaptics chips are designed to ensure that any such disruption results in a clean reset of the chip. The RMI protocol is designed to allow the system to recover gracefully in the event of such a spontaneous device reset. Together, these designs allow robust touch sensing even in ESD-prone or otherwise error prone environments, as described in section 2.7.3.

Spontaneous resets on the host side are also a fact of life in embedded systems. If the host restarts suddenly, it may not have had time to shut down the peripherals in an orderly way. An RMI device should be prepared to accept a “reset” command no matter what was happening earlier, even if a previous RMI physical layer transaction was interrupted partway through, and even if a special command or mode was already in progress on the device.

2.6. Register coherence

Depending on the needs of the individual RMI functions, groups of registers may be defined as being *coherent* for the purposes of reading and/or writing. There are many reasons why registers may need to be grouped into a coherent region. Typically, a particular set of data being reported may span more than one register. In that case, defining the group of registers to be coherent indicates that they belong to the same snapshot in time. For example, consider a timer that reports a 10-bit timer count in two RMI registers, the two most-significant bits in a register at address X, and the eight least-significant bits in another register at address X+1. Because the timer count may be incrementing while the registers are being read, the following sequence of events could occur:

1. Timer count is \$1FF.
2. Host reads the most significant Timer count register at addr X: \$01.
3. Timer count increments to \$200.
4. Host reads the least significant Timer count register at addr X+1: \$00.

At this point, the host reconstructs the timer count from the two register reads. Because the two reads were not coherent in time, the host will believe that the timer count is \$100 when the real timer count is \$200.

To solve these problems, RMI allows functions to define groups of registers as being *coherent regions*. These coherent regions are treated specially by the RMI device so that the information they either report (for reads) or accept (for writes) has the correct temporal significance. In the example above, an RMI device would treat the timer as a coherent region.

The coherent sequence of events would go as follows:

1. Timer count is \$1FF.
2. Host reads the most significant Timer count register from a coherent region at addr X: \$01. The RMI function saves all the registers belonging to this coherent region in a special buffer. The saved count is \$1FF.
3. Timer count increments to \$200.

4. Host reads the least significant Timer count register from a coherent region at addr $X+I$. Because the second read is also from the same coherent region, the RMI device reads from the special buffer instead of the current Timer Count: \$FF.

The host sees the value \$1FF, which represents the time at which the host started reading the Timer count.

2.6.1. Write access into coherent regions

A set of write accesses to a write-coherent region is said to be *committed* when the last register in the region is written. Writing the last register in a coherent region triggers an atomic update inside the device firmware for all of the written data belonging to the coherent region.

If a single transfer happens to write into two or more adjacent coherent regions, multiple commit operations will occur during the course of the transfer as the last register in each region is written.

Most write-coherency regions require that the host write every register in the region as part of every write access to the region. However, the individual RMI functions are free to define write access methods that permit a host to write a meaningful subset of the registers in their coherency regions. Even so, any permitted register subset must always include the last register in the region, otherwise the commit operation will not occur. These special access restrictions are defined in the function-specific sections of this document.

2.6.2. Read access into coherent regions

Read accesses into a coherent region are essentially ‘invisible’ to a host. During a read transfer, the first read access anywhere inside a coherent region cause a snapshot to be taken of all the registers that belong to the region. Subsequent register reads to the same region at sequentially increasing register addresses are serviced from the snapshot data, meaning that the subsequent register reads come from the same moment in time as the first read into the region.

A snapshot is invalidated (discarded) as soon as a register transfer occurs at any address other than a sequentially increasing address within the same region. This means that if a host repeatedly polls the same address within a region, a new snapshot is acquired for each poll operation. Once the host is satisfied with the results of the poll operation, subsequent reads at increasing addresses from within the coherent region are supplied from the final coherent snapshot. RMI functions typically place important status information that might be used for polling purposes at the start of a coherent region.

RMI functions that implement register read-access coherent regions describe their special access restrictions in the function-specific portions of this document.

2.7. Data reporting

An RMI device may have any number of *data sources*. In general, a data source corresponds to one independent sensor or input device, but in some cases a group of similar sensors (such as an array of buttons) are combined to form a single data source. A data source always defines one or more data registers.

2.7.1. Interrupt requests

RMI uses *interrupt requests* to signal whether or not a data source has information likely to be of interest to the host. Exactly what constitutes an interrupt, and when and at what rate new data will arrive, depend entirely on the kind of input device involved. The specification for each RMI function defines the exact rules for signaling interrupt requests for each of its data sources.

Each data source maintains an independent interrupt request state bit. The Interrupt Status register, described in section 3.2.2, reports the set of interrupt request bits in the device.

The Interrupt Request bit for a data source is ‘1’ if the data source has an interrupt request, or ‘0’ if there is no interrupt request for the source. Once a source has an interrupt request and its Interrupt Request bit has changed to ‘1’, the bit remains at ‘1’ until it is read, or (in some products) the host takes other actions that are documented to clear the interrupt request state of the data source.

The data registers always report meaningful data whether or not the interrupt request condition is present. Most often, an interrupt request state of ‘0’ implies that the data registers are unchanged since the last time the host read the data.

However, some data sources may define a more selective “interrupt” criterion, so that certain “insignificant” data changes may occur without causing interrupt request to be asserted. Very simple hosts could even read the data registers by periodic polling, observing the data but completely ignoring the Interrupt Request bits and the attention signal.

The post-reset default state of the bits in the Interrupt Status and Interrupt Enable registers is product-specific. Consult the product-specific documentation for details.

2.7.2. Attention signal

The host in an RMI system is in charge of all transactions with the device. When the device has an interrupt request to report to the host, it uses an *attention* mechanism to alert the host that it is time to read the data registers. Typically, a host system connects the attention signal to an interrupt input. The attention signal is merely advisory; the host is free to read the data registers at any time. The form of the attention signal depends on the physical interface; for example, see section 12.3.4 for a description of the attention signal for RMI-on-SPI.

Synaptics can supply RMI devices in which the attention signal is active-high or active-low. The attention polarity is a build-time option, and is not configurable at run-time.

The attention signal is said to be *asserted* if it is in the state that alerts the host (high for active-high polarity or low for active-low polarity). The attention signal is in the *de-asserted* state when it is not asserted.

The attention signaling model is compatible with both level-triggered and edge-triggered interrupt inputs on the host.

Every RMI device defines at least one Interrupt Status register that includes up to 8 Interrupt Enable bits, one bit per data source. For devices with more than 8 data sources, enough extra Interrupt Status registers are defined to hold all the Interrupt Enable bits defined by all the data sources present.

The attention signal is asserted whenever at least one interrupt source has a ‘1’ in its bit of the interrupt enable mask and its Interrupt Request bit is also ‘1’. The attention signal may become asserted or de-asserted if the host writes to the control registers to change the Interrupt Enable bits.

Note that most data sources continue to work, and continue to operate their Interrupt Request bits, even if their Interrupt Enable bit is ‘0’. The Interrupt Enable bit merely controls whether interrupt requests on a source will send an attention signal to the host.

The attention signal is de-asserted by reading all of the Interrupt Status registers in a device. This means that a host driver should process the interrupt handlers for all interrupt sources that are reporting ‘1’ when the Interrupt Status is read.

The host can disable attention by setting all the Interrupt Enable bits to ‘0’, thereby forcing the assertion signal to its de-asserted level. This gives the host a way to “disable interrupts” on the device side.

For example, in a system where several devices' attention pins have been logically OR'd together to drive a host interrupt pin, the host might wish to disable interrupts from some of the devices while remaining sensitive to other devices.

The attention signal is always asserted after device reset; see section 2.7.3.

2.7.3. Spontaneous resets

As noted in section 2.5.4, RMI is designed to be robust in the presence of spontaneous resets of RMI devices. Several properties of RMI are designed to work together to allow the host to detect reliably when the device has reset itself spontaneously:

- Every RMI function that implements an interrupt source can be configured to either assert or de-assert both its Interrupt Request and Interrupt Enable bits upon reset.
- Every RMI device contains a Device Control function, such as RMI Function \$01. An RMI Device Control function is always configured to assert both its Interrupt Request and Interrupt Enable bits upon reset.
- The assertion of any Interrupt Request and its corresponding Interrupt Enable causes the attention interrupt signal to become asserted after a reset.
- The attention signal prompts the host to read Interrupt Status register in order to determine the source of the interrupt.
- When the host reads the Interrupt Status register, it will find a '1' in the DevStatus interrupt request flag. This prompts the host to read the Device Status register.
- The Device Status register indicates that the device has lost its configuration due to a spontaneous reset.

The host should respond to a spontaneous device reset by fully reinitializing the device. Depending on the nature of the overall system, the host may even wish to use a spontaneous reset in one RMI device as a cue to reinitialize other parts of the system.

Hosts that operate the device in its default configuration, without ever writing to any control registers, will not necessarily be able to detect a spontaneous reset because the DevStatus interrupt request flag will always be '1'. However, these hosts need not be cognizant of spontaneous resets; from their point of view, the resetting device will simply pause briefly in its reporting of sensor activity and then resume normal operation.

3. Function \$01: RMI device control

Function \$01 implements a set of registers suitable as the foundation for controlling a large family of RMI products. Every RMI product requires a device control function to be present.

3.1. Function \$01: query registers

3.1.1. F01_RMI_Query0: Manufacturer ID query

This register reports the identity of the manufacturer of the RMI device. Synaptics RMI devices report a Manufacturer ID of \$01.

Name	7	6	5	4	3	2	1	0
F01_RMI_Query0	Manufacturer ID							

Figure 6. Function \$01 Manufacturer ID Query register

3.1.2. F01_RMI_Query1: Product Properties query

This byte contains bits that describe whether the RMI product has various optional properties.

Name	7	6	5	4	3	2	1	0
F01_RMI_Query1	Reserved			—	HasSensorID	—	NonCompliant	CustomMap

Figure 7. Function \$01 Product Properties query register

Each property bit is ‘1’ if the product has the associated property or ‘0’ if the product does not have the associated property. Reserved property bits report as ‘0’, but they may report as ‘1’ in devices that comply with a future version of RMI.

CustomMap (F01_RMI_Query1, bit 0)

When ‘1’, this bit indicates the presence of at least one custom, non RMI-compatible register in the register address map for this device. RMI defines no other information about the custom register implementation other than to flag its existence.

NonCompliant (F01_RMI_Query1, bit 1)

When ‘1’, this bit indicates that the device implements a register map that is not compliant with the RMI specification.

HasSensorID (F01_RMI_Query1, bit 3)

When ‘1’, this bit indicates that the Sensor ID query register (F01_RMI_Query22) exists.

The *CustomMap* and *NonCompliant* bits can be interpreted in conjunction with each other. For example, if *CustomMap* is set, and *NonCompliant* is clear, the device implements both a complete RMI-compliant register set in addition to one or more custom registers. If both *CustomMap* and *NonCompliant* are set, a device does not implement a fully RMI compliant register set, and it also implements one or more custom registers.

If *CustomMap* is clear, and *NonCompliant* is set, the device implements a non-compliant RMI register map that does not contain any custom registers. This might happen if a device were to implement a subset of RMI.

3.1.3. F01_RMI_Query2, 3: Product Info query

Standard Synaptics RMI products define the contents of these registers in their product-specific documentation. For custom products, the customer can define the meaning and contents of these registers when the product is ordered.

Name	7	6	5	4	3	2	1	0
F01_RMI_Query2	—				Product Info 0			
F01_RMI_Query3	—				Product Info 1			

Figure 8. Function \$01Product Info Query registers

3.1.4. F01_RMI_Query4 through 10: Device Serialization queries

These seven registers optionally record the individual identity of the device. Some RMI devices are not serialized at the factory; for unserialized devices, all of these registers report \$00.

Name	7	6	5	4	3	2	1	0
F01_RMI_Query4	—	—	—			Date Code Year		
F01_RMI_Query5	—	—	—	—		Date Code Month		
F01_RMI_Query6	—	—	—			Date Code Day		
F01_RMI_Query7	—				Tester ID (bits 13:7)			
F01_RMI_Query8	—				Tester ID (bits 6:0)			
F01_RMI_Query9	—				Serial Number (bits 13:7)			
F01_RMI_Query10	—				Serial Number (bits 6:0)			

Figure 9. Function \$01 Device Serialization Query registers

The various Device Serialization queries are defined as follows:

Date Code (*F01_RMI_Query4*, bits 4:0, *F01_RMI_Query5*, bits 3:0, *F01_RMI_Query6*, bits 4:0)

The date code registers are intended to record the date on which the module was manufactured. The actual interpretation is up to the manufacturer, but RMI diagnostic tools display this field assuming the bits are divided into year, month, and day fields as shown in Figure 9. The year code is a number from 1 to 31 to indicate years 2001–2032. The month code is a number from 1 to 12 to indicate the months January through December. The day code is a number from 1 to 31 to indicate the day of the month. The day field can be 0 to indicate “unknown day of the month;” the day and month fields can both be 0 to indicate “unknown day of the year;” the entire 16 bits can be zero to indicate “unknown manufacturing date.”

Tester ID (*F01_RMI_Query7*, bits 6:0, *F01_RMI_Query8*, bits 6:0)

The Tester ID registers are intended to identify the equipment used to manufacture or test the RMI device. The 14-bit Tester ID value \$0000 conventionally means “unknown tester.” The actual interpretation of these registers is up to the device manufacturer.

Serial Number (*F01_RMI_Query9*, bits 6:0, *F01_RMI_Query10*, bits 6:0)

The Serial Number registers are intended to record a unique serial number for a given tester and day. The 14-bit Serial Number value \$0000 conventionally means “no serial number recorded.”

The actual interpretation of these registers is up to the device manufacturer. RMI diagnostic tools display the tester ID and serial number in the form *tester:serial*, where *tester* and *serial* each are four hexadecimal digits.

If the Serial Number query registers (F01_RMI_Query9 and F01_RMI_Query10) are not \$0000, then the combination of Manufacturer ID, Product ID, Date Code, Tester ID, and Serial Number should be completely unique among all manufactured RMI devices. Nothing in the RMI standard itself uses the serialization information, but hosts and tools may rely on this uniqueness property when the Serial Number is non-zero.

3.1.5. F01_RMI_Query11 through 20: Product ID queries

These 10 bytes report the identity of the particular RMI device or product as an array of 7-bit ASCII characters.

Name	7	6	5	4	3	2	1	0
F01_RMI_Query11					Product ID, character 1			
F01_RMI_Query12					Product ID, character 2			
F01_RMI_Query13					Product ID, character 3			
⋮					⋮			
F01_RMI_Query18					Product ID, character 8			
F01_RMI_Query19					Product ID, character 9			
F01_RMI_Query20					Product ID, character 10			

Figure 10. Function \$01 Product ID Query registers

These registers form a string that identifies the product. Strings shorter than 10 characters always start at the first of these query registers (F01_RMI_Query11), and the unused characters at the end of the string will report as \$00.

The contents of the Product ID string is product-specific, and the actual contents will be described in the product-specific documentation.

3.1.6. F01_RMI_Query21: reserved for Synaptics use

This register is reserved for Synaptics use.

Name	7	6	5	4	3	2	1	0
F01_RMI_Query21					-----			

Figure 11. Function \$01 xxx query register (reserved)

3.1.7. F01_RMI_Query22: Sensor ID

This register exists only if HasSensorID (F01_RMI_Query1 bit3) is set to '1'.

The contents of this register identify the sensor attached to the device. The value is device-dependent and is described in the product-specific specification.

Name	7	6	5	4	3	2	1	0
F01_RMI_Query22								Sensor ID

Figure 12. Function \$01 Sensor ID query register

3.2. Function \$01: control registers

3.2.1. F01_RMI_Ctrl0: Device Control register

The Device Control register contains bits that control the pace of processing in the device, and the conditions under which it can enter low-power states.

Name	7	6	5	4	3	2	1	0
F01_RMI_Ctrl0	Configured	ReportRate	—	—	—	NoSleep	Sleep Mode	

Figure 13. Function \$01 Device Control register

The bits of this register are defined as follows:

Sleep Mode (F01_RMI_Ctrl0, bits 1:0)

This field controls power management on the device. This field affects all functions of the device together. Below are descriptions of all possible sleep modes.

Sleep Mode = '00': Normal Operation.

In this state, the device automatically and invisibly switches between full operation and a “doze” state in which finger sensing happens at a reduced rate (typically a few tens of milliseconds). The doze sensing rate is chosen so that almost any human finger action, such as rapid tapping, will still perform well.

This setting merely authorizes the device to doze when it is able. Products may be able to doze only under certain conditions, and may remain fully awake, for example, when a finger is present or when LED ramp animations are active.

Most RMI devices can be left in the Normal Operation setting at all times.

Sleep Mode = '01': Sensor Sleep.

This state fully disables touch sensors and similar “analog” inputs on the device. All touch sensors report the “not touched” state regardless of any finger presence. Digital inputs, such as mechanical buttons attached to GPI pins, continue to operate even in the Sensor Sleep state.

Setting the Sleep Mode field to Sensor Sleep constitutes a request to the device to enter the sleeping state. The device may continue to operate in a higher-power state for one or two more report periods before it goes to sleep.

All RMI devices support the Sensor Sleep state at least to the degree of forcing the touch sensors to the “not touched” state. In many devices, the Sensor Sleep state conserves additional power. In devices that do not support this deeper sleeping mode, the Sensor Sleep state is identical to the Normal Operation mode except for forcing the “not touched” state. Even then, this mode may help to conserve overall system power by interrupting the host less often.

Sleep Modes = '10, 11': Reserved.

These Sleep Mode encodings are reserved. Specific products may define sleep modes corresponding to these encodings which are particular to that product. Products implementing a reserved encoding describe its particular operational effects in their product-specific documentation.

The reset state of the Sleep Mode bits default to the Normal Operation state.

NoSleep (F01_RMI_Ctrl0, bit 2)

When set to '1', this bit disables whatever sleep mode may be selected by the Sleep Mode field, and forces the device to run at full power without sleeping. The reset state is '0', meaning that the current Sleep Mode is enabled.

Reserved (F01_RMI_Ctrl0, bits 5:3)

These bits are reserved for definition in future versions of the RMI protocol.

ReportRate (F01_RMI_Ctrl0, bit 6)

This field sets the report rate for the device. It applies in common to all functions on the device that have a natural report rate.

Many RMI functions divide time into *report periods* that occur at a *report rate*, roughly analogous to the “packet rate” of a mouse. If there are several such functions on an RMI device that work in terms of report periods, all the functions schedule their operation to the common report period of the device.

The encoding of the Report Rate field is largely device-dependent. The RMI standard does not require any particular encoding, although the value '0' corresponds to the device-preferred report rate. The reset value of the Report Rate field is '0'. If supported by a device, the value '1' will define a non-standard product-dependent report rate. RMI functions that work in terms of report periods assert interrupt requests, and therefore also the attention signal, at most once per report period.

Usually, report periods happen at a steady rate. Some conditions may cause the report period in progress to be canceled and a new report period started. This will not result in any loss of data, but it will add a visible irregularity to the steady rate of reports. For example, depending on the device implementation, writes to some control registers will cancel and restart the report period.

Some RMI functions do not schedule their activity in terms of report periods; these are known as *asynchronous* functions. Asynchronous RMI functions may assert an interrupt request on any schedule depending on the needs of the function. If an RMI device contains only asynchronous functions, its Report Rate field is unimplemented and resets to '0' (as described in section 2.2.1).

Configured (F01_RMI_Ctrl0, bit 7)

An RMI device may need to be configured (see section 2.2.2.1) after a device reset event.



Important: A host should write this bit to ‘1’ **before** beginning the configuration process in order to clear the ‘Unconfigured’ status bit in the Device Status register. Once the configuration is complete, the host should verify that the Unconfigured bit in the Device Status register is still reporting as ‘0’ (configured). This method guarantees that a host would properly detect a situation where an RMI device might unexpectedly reset during the configuration process for any reason.

Writing the Configured bit to ‘0’ has no effect. Reading the Configured bit always returns a ‘0’. The Unconfigured bit in the Device Status register always reports the actual configuration state of an RMI device.

3.2.2. F01_RMI_Ctrl1.*: Interrupt Enable register

The Interrupt Enable control register determines which interrupt sources are able to participate in the decision to assert the attention interrupt signal. Any function in an RMI device that defines interrupt sources is assigned bits in the Interrupt Enable register. If the RMI device defines more interrupt sources than will fit in a single Interrupt Enable register, a sufficient number of additional Interrupt Status registers are also defined at sequential addresses in the register map. Any unassigned Interrupt Enable bits in the register are treated as reserved bits. Every bit assigned to an Interrupt Enable register has a matching assignment made to the corresponding Interrupt Status register (see section 3.3.1).

Name	7	6	5	4	3	2	1	0
F01_RMI_Ctrl1.*	Int Enable 7	Int Enable 6	Int Enable 5	Int Enable 4	Int Enable 3	Int Enable 2	Int Enable 1	Int Enable 0

Figure 14. Function \$01 Interrupt Enable register

Each bit of this register controls whether the corresponding interrupt source asserts attention when it has an interrupt request. Bit n of this register is ‘1’ if the interrupt request on data source n should assert attention, or ‘0’ if the interrupt request on source n should not affect the attention signal. See section 2.7.2. Setting this field to all ‘0’ bits effectively disables the attention interrupt signal altogether.

Depending on the implementation, the effect on the attention signal of a change to the Interrupt Enable bits may be either immediate or deferred until the next report period.

Unimplemented bits in the Interrupt Enable register always report as ‘0’.

RMI Function \$01 is designed to typically assert an attention interrupt upon reset, as described in section 2.7.3. Under unusual circumstances, certain products may be configured so that the reset state of the bits in the Function \$01 Interrupt Enable register does not generate an interrupt; consult the product-specific documentation for details in those cases.

3.3. Function \$01: data registers

3.3.1. F01_RMI_Data0: Device Status register

The Device Status register reports events of interest to the host regarding the general status of the RMI device. Any change to the Device Status register causing it to become non-zero is indicated to the host by asserting the DevStat interrupt request bit in the Interrupt Status register.

Name	7	6	5	4	3	2	1	0
F01_RMI_Data0	Unconfigured	FlashProg	—	—	Status Code			

Figure 15. Function \$01 Device Status register

The bits of this register are defined as follows:

Status Code (F01_RMI_Data0, bits 3:0)

The Status Code field reports the most recent device status event. If several status conditions arise simultaneously, the actual status code reported is implementation-dependent.

Status conditions created by host actions, such as invalid configurations of control bits, may be reported instantly or they might not be reported until a few milliseconds after the offending register was written (for example, error conditions might not be checked until the next report period). Similarly, if new data is written to the control registers to correct the error condition, it may take a few milliseconds for the Status Code to clear.

RMI defines the following standard status codes:

Code \$00: No Error.

Code \$01: Reset occurred.

This code is reported if no other status event has occurred since the last time the device was reset. This error code is cleared when the Configured bit in control register 0 is written to '1' (see section 3.2.1).

Code \$02: Invalid Configuration.

This error signals a problem with the general configuration of the device, not specific to any one function. Many RMI devices do not implement this error.

Code \$03: Device Failure.

This error signals a hardware problem with the device, not specific to any one function. Many RMI devices do not implement this error. Other devices might, for example, signal this status code if the firmware fails a program memory self-check.

Products containing RMI function \$34 define the following status codes:

Code \$04: Configuration CRC Failure.

This error signals that the configuration of the device failed a program memory self-check and therefore normal operation of the device is not possible.

Code \$05: Firmware CRC Failure.

This error signals that the firmware failed a program memory self-check and therefore normal operation of the device is not possible.

Code \$06: CRC In Progress.

This error signals that the firmware is in the process of testing either the configuration area or the firmware area.

Codes \$07–\$0F: Reserved.

These status codes are reserved for definition by future versions of RMI.

Reserved (F01_RMI_Data0, bits 5:4)

These bits are reserved for definition in future versions of the RMI protocol.

FlashProg (F01_RMI_Data0, bit 6)

The *FlashProg* bit defines the current device operating mode. The *FlashProg* flag is set if the normal operation of the device is suspended because the device is in a Flash Programming enabled state.

This can be the result of the issuance of a Function \$34 Flash Program Enable command, or if there has been an error with the startup of the device that prevents its normal operation. The *Status Code* field can be read to give the reason for this bit being set. When in this mode the normal operation of the device is not possible; see section 10.1.2 for more information about this mode.

A ‘0’ indicates the device is operating in UI (User Interface) mode. A ‘1’ indicates the device is operating in Flash Programming mode, also known as *bootloader mode*. Devices that do not contain Function \$34 (Flash Programming) will always report ‘0’.

Unconfigured (F01_RMI_Data0, bit 7)

The Unconfigured flag reports as ‘1’ if the device loses its configuration for any reason. The Unconfigured flag can be cleared by writing the *Configured* bit of F01_RMI_Ctrl0 to a ‘1’.

3.3.2. F01_RMI_Data1.*: Interrupt Status register

The Interrupt Status register reports which of the set of possible interrupt sources are actively requesting interrupt service from the host. The RMI attention interrupt (see section 2.7.2) is asserted by the RMI device if any *Int Request* bit in the Interrupt Status register is ‘1’, and the corresponding *Int Enable* bit in the corresponding Interrupt Enable register is also ‘1’.

Name	7	6	5	4	3	2	1	0
F01_RMI_Data1.*	Int Request 7	Int Request 6	Int Request 5	Int Request 4	Int Request 3	Int Request 2	Int Request 1	Int Request 0

Figure 16. Function \$01 Interrupt Status register

Note: A ‘.*’ indicates a variable-sized register block. Every product contains at least one Interrupt Status register. The number of Interrupt Status registers implemented by an RMI device can be calculated by counting the total number of Interrupt Sources in the device:

$$\text{InterruptStatusRegisterCount} = \text{trunc}((\text{NumberOfInterruptSources} + 7) / 8)$$

Any function in an RMI device that defines interrupt sources is assigned bits in the Interrupt Status register. If the RMI device defines more interrupt sources than will fit in a single Interrupt Enable register, enough additional Interrupt Status registers are also defined by Function \$01 to hold the other *Int Enable* bits. If more than one Interrupt Status register is defined, the subsequent registers will be defined at sequential addresses after the first Interrupt Status register. The assignment of bits in the Interrupt Status registers to the set of interrupt sources in an RMI device is always identical to the assignment of the bits in the Interrupt Enable registers. This means that the number of Interrupt Status registers is always identical to the number of Interrupt Enable registers.

The act of reading an Interrupt Status register clears all the *Int Request* bits within it. The reset state of the implemented bits in the Interrupt Status register is product-specific.

Because every RMI device contains a Device Control function, and the Device Control function always defines one interrupt source (the DevStatus interrupt source), there will always be at least one Interrupt Status register in every RMI device.

3.3.3. Function \$01: interrupt source

The Data registers defined by with Function \$01 are associated with a single interrupt source, called the *DevStatus* interrupt. The DevStatus interrupt request is asserted whenever the RMI device has experienced some unusual event that requires the host's immediate attention.

Any product that includes Function \$01 allocates a DevStatus interrupt request bit in the Interrupt Status register (see section 3.3.2), and a DevStatus interrupt enable bit in the Interrupt Enable register (see section 3.2.2). The position of the allocated interrupt bit within those registers is product-specific; consult the product-specific documentation to determine their location.

The reset state of the DevStatus interrupt request bit and the DevStatus interrupt enable bit is product-specific. However, in a typical device, the default reset state for both of these bits is '1'. This means that after a reset, a DevStatus interrupt will be pending in the Interrupt Status register, and enabled in the Interrupt Enable register, thus asserting the host attention interrupt. This ensures that any reset event will assert the DevStatus interrupt to report a loss of configuration.

3.4. Function \$01: command registers

3.4.1. F01_RMI_Cmd0: Device Command register

The Device Command register is used to issue special commands to an RMI device. For general guidelines on the use and operation of command registers, see section 2.2.4.

Name	7	6	5	4	3	2	1	0
F01_RMI_Cmd0	—	—	—	—	—	—	—	Reset

Figure 17. Function \$01 Device Command register

The bits of this register are defined as follows:

Reset command (F01_RMI_Cmd0, bit 0)

Writing a ‘1’ to this bit causes the device to reset exactly as if its RESET pin had been pulled low.

The device’s host interface (SMBus or SPI pins) may not operate for a certain amount of time T_{RESC} (about 1 ms) after a reset command or a low level on the RESET pin. This delay is much shorter than the delay T_{POR} before the host interface begins operating after a power-on reset. The T_{RESC} delay begins at the end of the write transaction that writes to this register (for example, at the rise of SSB in the case of RMI on SPI).

Note: The device asserts attention as soon as it is capable of responding to an operation on its physical interface.

Reserved (F01_RMI_Cmd0, bits 7:1)

These command bits are *reserved*.

4. Function \$08: BIST

Function \$08 implements controls for the BIST (Built-In Self Test) functions for testing, characterization, and analysis of a system and its ASIC. There are two BIST functions, Function \$08 and Function \$09. Only one of these functions is applicable for a device; it is dependent upon the sensor chip. See the device Product Specification for information on which BIST function is used.

4.1. Typical BIST usage scenario

The BIST function is typically used to test devices after assembly. The basic functionality is not intended as a complete test or replacement for module testing. BIST can be run after the device has powered on and begun reporting after POR (PowerOnReset).

The BIST control limit registers default to the correct values for the test, but they can be changed by re-flashing the device with new Firmware.

The command RunBIST (F08_BIST_Cmd0, bit 0) executes TestNumber=0 by default, runs a complete BIST, and reports the first failing sub-test (for example, *Test1*).

In a typical case, running the default BIST command should result in a Passing (\$00) result in the F08_BIST_Data1 register. In the case of a failing non-zero result, the output of the data register indicates the failed sub-test number. The failing sub-test result can be loaded into the Test Number Control register F08_BIST_Data0 to determine the cause of the fault. Running RunBIST (F08_BIST_Cmd0) again, with the Test Number Control register properly loaded with the sub-test number, produces a result in the BIST data registers that indicates an electrode that failed the BIST sub-test.

4.2. Function \$08: query registers

The various BIST queries provide the mechanism for configuring and reading the BIST controls (limits), commands (tests), and data (results) variables. The BIST limits may be configurable and set to defaults by Flash-backed registers or set in Flash on a per-product basis. Additional control limit registers are necessary for additional BIST tests.

Name	7	6	5	4	3	2	1	0
F08_BIST_Query0	Limit Register Count							
F08_BIST_Query1	HostTestEn	—	—	—	—	—	—	—

Figure 18. Function \$08 query registers

The locations of the control, data, and command registers for basic BIST tests are calculated from the F08_BIST_Query0 register. Any additional BIST tests and adjustments to the BIST register placements can be calculated based on F08_BIST_Query1.

Limit Register Count (F08_BIST_Query0)

Indicates the number of control registers used to set the limits for BIST testing capabilities.

HostTestEn (F08_BIST_Query1, bit 7)

Indicates that BIST host-level testing is available.

4.3. Function \$08: control registers

The control registers determine the limits for each of the implemented BIST tests. Depending on the configuration, a device can have one or two defined tests. The example below describes a Function \$08 configuration with two tests:

Name	7	6	5	4	3	2	1	0
F08_BIST_Ctrl0					Test1LimitLo			
F08_BIST_Ctrl1					Test1LimitHi			
F08_BIST_Ctrl2					Test1LimitDiff			
F08_BIST_Ctrl3					Test2LimitLo			
F08_BIST_Ctrl4					Test2LimitHi			
F08_BIST_Ctrl5					Test2LimitDiff			

Figure 19. Function \$08 Limit control registers

These registers are defined as follows:

Test1LimitLo (F08_BIST_Ctrl0)

Controls the acceptance Low BIST Limit for Test1.

Test1LimitHi (F08_BIST_Ctrl1)

Controls the acceptance High BIST Limit for Test1.

Test1LimitDiff (F08_BIST_Ctrl2)

Controls the acceptance Difference BIST Limit for Test1.

Test2LimitLo (F08_BIST_Ctrl3)

Controls the acceptance Low BIST Limit for Test2.

Test2LimitHi (F08_BIST_Ctrl4)

Controls the acceptance High BIST Limit for Test2.

Test2LimitDiff (F08_BIST_Ctrl5)

Controls the acceptance Difference BIST Limit for Test2.

4.4. Function \$08: data registers

Function \$08's data result registers are allocated as necessary for the tests defined by design capability. The data registers are filled upon the completion of a command. Reported data values may change, depending on the configuration of the control limit and the command executed. Future unimplemented extended BIST capabilities may add additional data registers.

The Test Number Control register determines which BIST tests are run. If '0', all implemented tests are run.

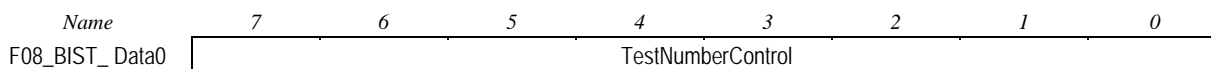


Figure 20. Function \$08 Test Number Control data register

TestNumberControl (F08_BIST_Data0)

Controls the test (or tests) run by the BIST command. Test capabilities are defined and documented on a design basis. Additional tests may be available, but not all devices will have all tests or necessarily sequential test numbers. '3'–'255' are reserved test numbers. By default:

- If '0', the complete BIST test executes each of the one or more BIST sub-tests in order until there is a failure. Reports the first failing test in the Test Result data register.
- If '1', then the RunBIST (F08_BIST_Cmd0) command executes the BIST sub-test 1 using the limits set by the Test1 limit control registers. Reports the failing electrode in the Test Result data register.
- If '2', then the RunBIST (F08_BIST_Cmd0) command executes the BIST sub-test 2 using the limits set by the Test2 limit control registers. Reports the failing electrode in the Test Result data register.

The Overall BIST Result register is '0' for success, or indicates the number of the first failing test.

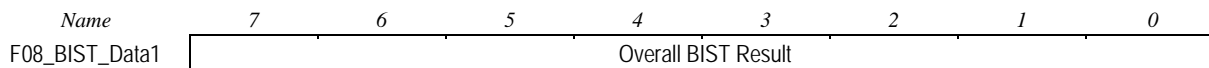


Figure 21. Function \$08 Overall BIST Result data register

Overall BIST Result (F08_BIST_Data1)

Reports the overall result of the BIST command executed:

- If '0', all requested BIST tests have passed.
- If '1', Test 1 has failed and F08_BIST_Data2 will indicate the failing electrode number.
- If '2', Test 2 has failed and F08_BIST_Data2 will indicate the failing electrode number.

'3-255' are reserved test numbers.

If the Overall BIST Result is non-zero, then the Test Result register contains test specific error information or the default complete test.

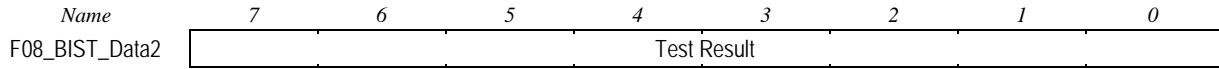


Figure 22. Function \$08 Test Result data register

For example, if Test1 failed, the Test Result data register would indicate which limit test failed and an electrode number associated with the failure, as shown below:

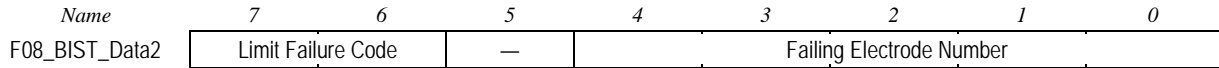


Figure 23. Function \$08 Test Result (Test1)

Failing Electrode Number (F08_BIST_Data2, bits 4:0)

Reports an electrode that fails the test. The value of this register is meaningful only if F08_BIST_Data1 is non-zero.

Limit Failure Code (F08_BIST_Data2, bits 7:6)

Reports which limit test fails the test:

00 = No failure.

01 = Low Limit failed.

10 = High Limit failed.

11 = Difference Limit failed.

4.4.1. Function \$08: interrupt source

The BIST data source asserts an interrupt request at the end of any report period in which a command register bit has been cleared.

4.5. Function \$08: command registers

The BIST command register provides the mechanism for executing the built-in self tests. Test times vary; test completion is indicated by the automatic clearing of the *RunBIST* bit and the assertion of the BIST interrupt request.

Name	7	6	5	4	3	2	1	0
F08_BIST_Cmd0	—	—	—	—	—	—	—	RunBIST

Figure 24. Function \$08 command register

RunBIST (*F08_BIST_Cmd0*, bit 0)

Runs the BIST using the test number and limits set in the data and control registers.

5. Function \$09: BIST

Function \$09 implements controls for the BIST (Built-In Self Test) functions for testing, characterization, and analysis of a system and its ASIC. There are two BIST functions, Function \$08 and Function \$09. Only one of these functions is applicable for a device; it is dependent upon the sensor chip. See the device Product Specification for information on which BIST function is used.

5.1. Typical BIST usage scenario

The BIST function is typically used to test devices after assembly. The basic functionality is not intended as a complete test or replacement for module testing. BIST can be run after the device has powered on and begun reporting after POR (PowerOnReset).

The BIST control limit registers default to the correct values for the test, but they can be changed by re-flashing the device with new firmware.

The command RunBIST (F09_BIST_Cmd0 bit 0) executes TestNumber=0 by default, runs a complete BIST, and reports the first failing sub-test (for example, Test1).

In a typical case, running the default BIST command should result in a Passing (\$00) result in the F09_BIST_Data1 register. In the case of a failing non-zero result, the output of the data register indicates the failed sub-test number. The failing sub-test result can be loaded into the Test Number Control register F09_BIST_Data0 to determine the cause of the fault. Running RunBIST (F09_BIST_Cmd0) again, with the Test Number Control register properly loaded with the sub-test number, produces a result in the BIST data registers that indicates an electrode that failed the BIST sub-test.

5.2. Function \$09: query registers

The various BIST queries provide the mechanism for configuring and reading the BIST controls (limits), commands (tests), and data (results) variables. The BIST limits may be configurable and set to defaults by flash-backed registers or set in flash on a per-product basis. Additional control limit registers are necessary for additional BIST tests.

Name	7	6	5	4	3	2	1	0
F09_BIST_Query0	Limit Register Count							
F09_BIST_Query1	HostTestEn	InternalLimits	—	—	—	Result Register Count		

Figure 25. Function \$09 query registers

The locations of the control, data, and command registers for basic BIST tests are calculated from the F09_BIST_Query0 register. Any additional BIST tests and adjustments to the BIST register placements can be calculated based on F09_BIST_Query1.

Limit Register Count (F09_BIST_Query0)

Indicates the number of control registers used to set the limits for BIST testing capabilities.

Result Register Count (F09_BIST_Query1, bits 2:0)

Indicates the number of result registers used to report BIST test results.

InternalLimits (F09_BIST_Query1, bit 6)

Indicates that BIST internal limits are available.

HostTestEn (F09_BIST_Query1, bit 7)

Indicates that BIST host-level testing is available. Test Limit control registers (for example, F09_BIST_Ctrl0 through 5) can be updated by the host before the BIST command is run and those values will be used to determine failures.

5.3. Function \$09: control registers

The control registers determine the limits for each of the implemented BIST commands.

Name	7	6	5	4	3	2	1	0
F09_BIST_Ctrl0					Test1LimitLo			
F09_BIST_Ctrl1					Test1LimitHi			
F09_BIST_Ctrl2					Test1LimitDiff			
F09_BIST_Ctrl3					Test2LimitLo			
F09_BIST_Ctrl4					Test2LimitHi			
F09_BIST_Ctrl5					Test2LimitDiff			

Figure 26. Function \$09 Limit control registers

These registers are defined as follows:

Test1LimitLo (F09_BIST_Ctrl0)

Controls the acceptance Low BIST Limit for Test1.

Test1LimitHi (F09_BIST_Ctrl1)

Controls the acceptance High BIST Limit for Test1.

Test1LimitDiff (F09_BIST_Ctrl2)

Controls the acceptance Difference BIST Limit for Test1.

Test2LimitLo (F09_BIST_Ctrl3)

Controls the acceptance Low BIST Limit for Test2.

Test2LimitHi (F09_BIST_Ctrl4)

Controls the acceptance High BIST Limit for Test2.

Test2LimitDiff (F09_BIST_Ctrl5)

Controls the acceptance Difference BIST Limit for Test2.

5.4. Function \$09: data registers

Function \$09's data result registers are allocated as necessary for the tests defined by design capability. The data registers are filled upon the completion of a command. Reported data values may change, depending on the configuration of the control limit and the command executed. Future unimplemented extended BIST capabilities may add additional data registers. The number of Test Result data registers is indicated in the *ResultRegisterCount* (F09_BIST_Query1, bits 2:0) field.

The Test Number Control register determines which BIST tests are run. If '0', all implemented tests are run.

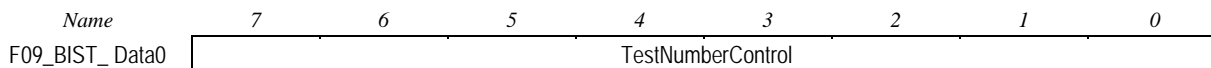


Figure 27. Function \$09 Test Number Control data register

TestNumberControl (F09_BIST_Data0)

Controls the test (or tests) run by the BIST command. Test capabilities are defined and documented on a design basis. Additional tests may be available (beyond 0, 1 and 2), but not all devices will have all tests or necessarily sequential test numbers. '3'–'255' are reserved test numbers. By default:

- If '0', the complete BIST test executes each of the one or more BIST sub-tests in order until there is a failure. Reports the first failing test in the Test Result data register.
- If '1', then the RunBIST (F09_BIST_Cmd0) command executes the BIST sub-test 1 using the limits set by the Test1Limit control registers.
Reports the failing electrodes in the Test Result data registers.
- If '2', then the RunBIST (F09_BIST_Cmd0) command executes the BIST sub-test 2 using the limits set by the Test2Limit control registers. Reports the failing electrodes in the Test Result data registers.

The Overall BIST Result register is '0' for success, or indicates the number of the first failing test.

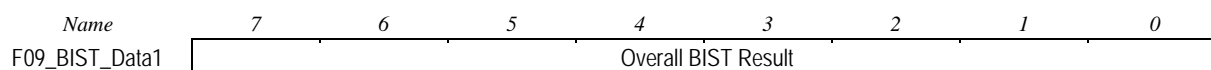


Figure 28. Function \$09 Overall BIST Result data register

Overall BIST Result (F09_BIST_Data1)

Reports the overall result of the BIST command executed:

- If '0', all requested BIST tests have passed.
- If '1', Test 1 has failed and F09_BIST_Data2 and F09_BIST_Data3 indicate the failing electrode numbers.
- If '2', Test 2 has failed and F09_BIST_Data2 and F09_BIST_Data3 indicate the failing electrode numbers.

'3-255' are reserved test numbers.

When *ResultRegisterCount* (F09_BIST_Query1 bits 2:0) field is 2, multiple data registers provide the failure results:

Name	7	6	5	4	3	2	1	0
F09_BIST_Data2	Test Result1							
F09_BIST_Data3	Test Result2							

Figure 29. Function \$09 Test Results

For example, if Test1 failed, the Test Result data registers would indicate which limit test failed and the corresponding transmitter and receiver electrodes associated with the failure, as shown below:

Name	7	6	5	4	3	2	1	0
F09_BIST_Data2	Limit Failure Code		Receiver Number					
F09_BIST_Data3	Transmitter Number							

Figure 30. Function \$09 Test Results, Test1 example

Receiver Number (F09_BIST_Data2, bits 5:0)

Reports an electrode that fails the test. The value of this register is meaningful only if F09_BIST_Data1 is non-zero. Indicates the first failing receiver number.

Limit Failure Code (F09_BIST_Data2, bits 7:6)

Reports which limit test fails:

‘00’ = No failure.

‘01’ = Low Limit failed.

‘10’ = High Limit failed.

‘11’ = Difference Limit failed.

Transmitter Number (F09_BIST_Data2)

Reports an electrode that fails the test. The value of this register is meaningful only if F09_BIST_Data1 is non-zero. Indicates the first failing transmitter number.

5.4.1. Function \$09: interrupt source

The BIST data source asserts an interrupt request at the end of any report period in which a command register bit has been cleared.

5.5. Function \$09: command registers

The BIST command register provides a mechanism for executing the built-in self tests. Test times vary; test completion is indicated by the automatic clearing of the *RunBIST* bit and the assertion of the BIST interrupt request.

Name	7	6	5	4	3	2	1	0
F09_BIST_Cmd0	—	—	—	—	—	—	—	RunBIST

Figure 31. Function \$09 command register

RunBIST (*F09_BIST_Cmd0*)

Runs the BIST using the test number and limits set in the data and control registers.

6. Function \$11: 2-D sensors

Function \$11 implements two-dimensional touch position sensors, such as the Synaptics TouchPad™ or ClearPad™ products. RMI4 has three potential data sources for 2-D devices: absolute, relative, and gestures:

- The absolute data source reports the positions of one or more fingers in the native X-Y coordinates of the 2-D sensor.
- The relative data source reports the delta X-Y coordinates.
- The gesture data source reports detection of finger gestures, and shares an interrupt request bit and an interrupt enable bit with the absolute data source.

6.1. Number of 2-D sensors

The Number of Sensors field of the F11_2D_Properties register expresses how many distinct 2-D sensors are present in the device. When more than one 2-D sensor is present in the device, each operates independently of the other. Each sensor reports its data in separate data registers with separate interrupt request and interrupt enable bits. Similarly each 2-D sensor's properties are described by a separate set of queries, and each is configured by a separate set of control registers.

The grouping of query, control, data and command registers in the register map for multiple 2-D sensors is analogous to the grouping of RMI functions. For example, the data registers for the first 2-D sensor is followed by the data registers for the next. The order of the grouping of multiple sensors can't be changed.

6.2. Function \$11: query registers

A device may contain more than one distinct 2-D sensor. As a result, there are two classes of Function \$11 queries:

1. Queries that apply to all 2-D sensors in a device.
2. Queries that apply to a particular 2-D sensor within a device.

The per-device 2-D query registers are placed first in the register map, followed by one block of per-sensor query registers for each 2-D sensor supported by the device.

6.2.1. F11_2D_Query0: per-device query registers

RMI Function \$11 defines a single per-device query register.

Name	7	6	5	4	3	2	1	0
F11_2D_Query0	—	—	—	—	—	NumberOfSensors		

Figure 32. Function \$11 Per-device query register

The fields in the register are defined as follows:

NumberOfSensors (F11_2D_Query0, bits 2:0)

This 3-bit field describes the number of 2-D sensors supported by Function \$11. It is zero-based, so a value of ‘0’ means one sensor, ‘1’ means two sensors, and so on.

Note: The number of sensors indicated by this field influences the size and layout of a particular device’s Function \$11 register map.

6.2.2. F11_2D_Query1 through 10: per-sensor query registers

Each 2-D sensor has its own set of query registers, used to describe its own particular characteristics.

Name	7	6	5	4	3	2	1	0
F11_2D_Query1	Configurable	HasSensitivityAdjust	HasGestures	HasAbs	HasRel	NumberOfFingers		
F11_2D_Query2	—	Number of X Electrodes						
F11_2D_Query3	—	Number of Y Electrodes						
F11_2D_Query4	—	Maximum Electrodes						
F11_2D_Query5	—	—	—	HasDribble	HasAdjHyst	HasAnchoredFinger	Abs Data Size	
F11_2D_Query6	—	—	—	—	—	—	—	—
F11_2D_Query7	—	HasPinch	HasPress	HasFlick	HasEarlyTap	HasDoubleTap	HasTapandHold	HasSingleTap
F11_2D_Query8	—	—	HasMultiFingerScroll	IndividualScrollZones	HasScrollZones	HasTouchShapes	HasRotate	HasPalmDet
F11_2D_Query9	—	—	—	—	—	—	—	—
F11_2D_Query10	—	—	—	Number of TouchShapes				

Figure 33. Function \$11 Per-sensor query registers

The fields in these registers are defined in the following sections.

6.2.2.1. F11_2D_Query1: general sensor information

NumberOfFingers (F11_2D_Query1, bits 2:0)

This 3-bit field describes the maximum number of fingers the 2-D sensor supports:

Table 2

Encoding	Maximum Number of Fingers Reported
000	1
001	2
010	3
011	4
100	5
101	10
110	Reserved
111	Reserved

Note: Because the field defines a **maximum** finger count, a 10-finger maximum still lets the device report any number of fingers up to and including 10.

HasRel (F11_2D_Query1, bit 3)

If *HasRel* reports as ‘1’, then the sensor supports relative mode, and the relative mode data registers are present in the register map. The relative mode data registers indicate finger movement in the form of position deltas since the last report.

HasAbs (F11_2D_Query1, bit 4)

If *HasAbs* reports as '1', then the sensor supports absolute mode, and absolute mode data registers are present in the register map. The absolute mode data registers provide the finger positions in the form of absolute positions.

HasGestures (F11_2D_Query1, bit 5)

If *HasGestures* reports as '1', then the sensor supports gesture processing. The gesture data source interprets specific finger movement events over short periods of time to provide the host with information regarding higher-level user inputs such as tapping, pinching, flicking, or pressing.

HasSensitivityAdjust (F11_2D_Query1, bit 6)

If the *HasSensitivityAdjust* bit or the *HasAdjHyst* bit (F11_2D_Query5, bit 3) report as '1', then the sensor supports a global sensitivity adjustment. This allows a host to make small adjustments to the overall sensitivity of the pad. If this bit reports as '1', then register F11_2D_Ctrl14 (Sensitivity Adjust) will appear in the register map; otherwise, it will not be present.

Configurable (F11_2D_Query1, bit 7)

If '0', the sensor mapping control registers are not configurable at run-time:

- *MaximumElectrodes* is the sum of *NumberOfXElectrodes* and *NumberOfYElectrodes*.
- The sensor map control registers are read-only.
- The sensitivity control registers are read-only.

If '1', the sensor mappings are configurable at run-time:

- *NumberOfXElectrodes* and *NumberOfYElectrodes* represent the maximum number of X and Y electrodes supported.
- The sensor map control registers are read-write.

6.2.2.2. F11_2D_Query2,3: number of X and Y electrodes*NumberOfXElectrodes (F11_2D_Query2, bits 6:0)**NumberOfYElectrodes (F11_2D_Query3, bits 6:0)*

If the *Configurable* bit is '1', then the *NumberOfXElectrodes* and *NumberOfYElectrodes* queries describe the maximum number of electrodes the 2-D sensor supports on the corresponding axis. The sum of these can't exceed *MaximumElectrodes*.

If the *Configurable* bit is '0', then the *NumberOfXElectrodes* and *NumberOfYElectrodes* queries describe the exact number of electrodes on each axis.

6.2.2.3. F11_2D_Query4: maximum electrodes*MaximumElectrodes (F11_2D_Query4, bits 6:0)*

MaximumElectrodes specifies the size of the SensorMap control register arrays. For configurable builds, this register also describes the maximum number of electrodes that the 2-D sensor supports. For non-configurable builds, '0' is a valid value for this register, even if F11_2D_Query2 and F11_2D_Query3 are non-zero.

6.2.2.4. F11_2D_Query5: absolute data source

This query register is present only if *hasAbs* in F11_2D_Query1 reports as '1'.

AbsDataSize (F11_2D_Query5 bits 1:0)

This two bit field describes the type of data reported by the absolute data source. From this, the total number of absolute data registers can be calculated:

- '00' – The absolute position data source will report X, Y, Z and W data using a total of five data registers. This block of registers will be replicated for each finger present in the sensor.
- '01', '10', '11' – *Reserved*.

HasAnchoredFinger (F11_2D_Query5, bit 2)

If *HasAnchoredFinger* reports as '1', then the sensor supports the high-precision second finger tracking provided by the F11_2D_Ctrl1 manual tracking and motion sensitivity options. This is a two finger mode, and is only available when the *numberOfFingers* query reports that two or more fingers are supported. If a first finger touches and remains steady, then a second finger can be tracked with high precision. If the first finger moves, then the reported positions of both the first and second fingers may degrade.

HasAdjHyst (F11_2D_Query5, bit 3)

If *HasAdjHyst* reports as '1', the sensor's Z hysteresis can be changed. If Z falls below the release threshold (the smaller of the two values), it is interpreted as a finger not touching the sensor; if Z rises above the touch threshold, it is interpreted as a finger touching the sensor.

If the *HasAdjHyst* bit or the *HasSensitivityAdjust* bit (F11_2D_Query1, bit 6) report as '1', then the *Sensitivity Adjust* register (F11_2D_Ctrl14) will appear in the register map; otherwise, it will not be present.

HasDribble (F11_2D_Query5, bit 4)

If *HasDribble* reports as '1', the sensor supports the generation of dribble interrupts, which may be enabled or disabled with the Dribble bit (F11_2D_Ctrl0, bit 6).

Typically, interrupts are generated (ATTN is asserted) when a finger arrives, lifts, or moves. The exact criteria for generating interrupts varies from one device to the next and is also usually configurable by the host (for example, some devices are configured to generate interrupts whenever a finger is present, while others generate interrupts only while the finger is present and moving).

Interrupts are generated only while the criteria are met. For example, if the device is configured to generate interrupts only while the finger is moving, the device stops generating interrupts the instant the finger stops moving.

When dribble is enabled, interrupts continue to be generated for a short while even *after* the interrupt-generating criteria are no longer met. These extra interrupts are called dribble interrupts.

6.2.2.5. *F11_2D_Query6: relative data source*

This query register is present only if *HasRel* in *F11_2D_Query1* reports as '1'. This register currently defines no queries regarding the relative data source.

6.2.2.6. *F11_2D_Query7, 8: gesture information*

These query registers are present only if *HasGestures* in *F11_2D_Query1* reports as '1'.

HasSingleTap (F11_2D_Query7, bit 0)

If *HasSingleTap* reports as '1', then a basic *single-tap* gesture is supported. A single-tap is defined to be a sequence of events where a finger lands on the sensor, stays in essentially the same location (defined by *Maximum Tap Distance*) for no more than a brief period of time (defined by *Maximum Tap Time*), and then leaves.

If *HasDoubleTap* or *HasTapAndHold* is true, a single-tap will not be reported for a short period of time after the finger leaves. The delay is necessary to distinguish a single-tap from the start of a tap-and-hold or double-tap gesture. If a host application desires to act on a tap event as soon as the finger lifts, it should use the early-tap report instead.

HasTapAndHold (F11_2D_Query7, bit 1)

If *HasTapAndHold* reports as '1', then a basic *tap-and-hold* gesture is supported. A tap-and-hold is defined to be a sequence of events where a finger lands on the sensor, stays in essentially the same location (defined by *Maximum Tap Distance*) for no more than a brief period of time (defined by *Maximum Tap Time*), leaves the sensor, then lands again in the same approximate location within a brief period time (a function of *Maximum Tap Time*), and remains touching the sensor for longer than *Maximum Tap Time*.

HasDoubleTap (F11_2D_Query7, bit 2)

If *HasDoubleTap* reports as '1', then a *double-tap* gesture is supported. A double-tap is defined to be a sequence of two taps separated by a brief time interval based on *Maximum Tap Time*. A double-tap is reported as soon as the finger leaves the sensor for the second time.

HasEarlyTap (F11_2D_Query7, bit 3)

If *HasEarlyTap* reports as '1', then *early taps* are reported by the gesture data source. An early tap is reported as soon as the finger lifts for any tap event that could be interpreted as either a single tap or as the first tap of a double-tap or tap-and-hold gesture.

HasFlick (F11_2D_Query7, bit 4)

If *HasFlick* reports as '1', then the 2-D sensor supports a *flick* gesture. A flick gesture is defined to be where a single finger leaves the sensor while moving faster than a specified speed (defined by *Minimum Flick Speed*) and beyond a specified distance (defined by *Minimum Flick Distance*).

HasPress (F11_2D_Query7, bit 5)

If *HasPress* reports as '1', then the *press* gesture is supported. In a press gesture, a single finger touches the sensor and stays in essentially the same location for a moderate interval of time (defined by *Minimum Press Time*).

HasPinch (F11_2D_Query7, bit 6)

If *HasPinch* reports as '1', then the sensor supports the two-finger *pinch* gesture. A pinch gesture is defined to be two fingers touching the pad and moving either towards each other or away from each other.

HasPalmDet (F11_2D_Query8, bit 0)

If *HasPalmDet* reports as ‘1’, then the 2-D sensor notifies the host whenever a large conductive object such as a palm or a cheek touches the 2-D sensor.

Note: Most 2-D data reporting is inhibited while a palm is detected.

HasRotate (F11_2D_Query8, bit 1)

If *HasRotate* reports as ‘1’, then the sensor supports the two finger *rotate* gesture. A rotate gesture is defined as:

- exactly two fingers touching the pad,
- the two fingers maintaining approximately the same distance apart (if the fingers move together, the sensor may interpret the movement as a pinch gesture), and
- the fingers swiveling clockwise or counterclockwise on the sensor surface, as if the hand is turning a dial or knob.

HasTouchShapes (F11_2D_Query8, bit 2)

If *HasTouchShapes* reports as ‘1’, then *TouchShapes* are supported. A *TouchShape* is a fixed rectangular area on the sensor that behaves like a capacitive button.

If *HasTouchShapes* reports as ‘1’, then the *F11_2D_Query10* register is used by this device.

HasScrollZones (F11_2D_Query8, bit 3)

If *HasScrollZones* reports as ‘1’, then *ScrollZones* are supported. A *ScrollZone* is a narrow rectangular area on one of the four edges of the sensor that reports relative motion. The zones parallel to the Y axis are called “Y Left” (at the edge of the screen where X is at a minimum) and “Y Right” (at the edge where X is at a maximum). The zones parallel to the X axis are called “X Lower” (at the edge where Y is at a minimum) and “X Upper” (at the edge where Y is at a maximum).

IndividualScrollZones (F11_2D_Query8, bit 4)

If *IndividualScrollZones* reports as ‘1’, then there are four Scroll Motion data registers, one for each of the four ScrollZones. If *IndividualScrollZones* reports as ‘0’, then there are only two Scroll Motion data registers. Motion on either of the “X” ScrollZones is reported in the “X Lower Scroll Motion” data register, and motion on either of the “Y” ScrollZones is reported in the “Y Right Scroll Motion” data register.

HasMultiFingerScroll (F11_2D_Query8, bit 5)

If *HasMultiFingerScroll* reports as ‘1’, then MultiFinger Scrolling is supported. When multiple fingers land at the same time and move in unison, the MultiFinger Scrolling gesture bit is set and the amount of scrolling is reported.

6.2.2.7. *F11_2D_Query10: TouchShape support*

This register is used for TouchShape support. It only exists if *HasTouchShapes* (F11_2D_Query8, bit 2) is set to ‘1’.

NumberOfTouchShapes (F11_2D_Query10, bits 4:0)

Holds the number (minus one) of TouchShapes supported by the sensor.

6.3. Function \$11: control registers

These registers control the operation of the 2-D sensors.

Name	7	6	5	4	3	2	1	0
F11_2D_Ctrl0	—	Dribble	RelBallistics	RelPosFilt	AbsPosFilt	ReportingMode		
F11_2D_Ctrl1	ManTracked Finger	ManTrackEn	MotionSensitivity		PalmDetectThreshold			
F11_2D_Ctrl2	DeltaXPosThreshold							
F11_2D_Ctrl3	DeltaYPosThreshold							
F11_2D_Ctrl4	Velocity							
F11_2D_Ctrl5	Acceleration							
F11_2D_Ctrl6	SensorMaxXPos 7:0							
F11_2D_Ctrl7	—	—	—	—	SensorMaxXPos 11:8			
F11_2D_Ctrl8	SensorMaxYPos 7:0							
F11_2D_Ctrl9	—	—	—	—	SensorMaxYPos 11:8			
F11_2D_Ctrl10	—	Pinch Interrupt Enable	Press Interrupt Enable	Flick Interrupt Enable	Early Tap Interrupt Enable	DoubleTap Interrupt Enable	TapAndHold Interrupt Enable	Single Tap Interrupt Enable
F11_2D_Ctrl11	—	—	—	MultiFinger Scroll Interrupt Enable	ScrollZone Interrupt Enable	TouchShape Interrupt Enable	Rotate Interrupt Enable	PalmDetect Interrupt Enable
F11_2D_Ctrl12.*	XYSel	SensorMap						
F11_2D_Ctrl13.*	Reserved							
F11_2D_Ctrl14	Hysteresis Adjustment			Sensitivity Adjustment				
F11_2D_Ctrl15	Maximum Tap Time							
F11_2D_Ctrl16	Minimum Press Time							
F11_2D_Ctrl17	Maximum Tap Distance							
F11_2D_Ctrl18	Minimum Flick Distance							
F11_2D_Ctrl19	Minimum Flick Speed							

Figure 34. Function \$11 control registers

6.3.1. F11_2D_Ctrl0: general control

This register contains general control bits that affect 2-D reporting and operation. The fields are defined as follows:

ReportingMode (F11_2D_Ctrl0, bits 2:0)

Depending on the specific application, a host may desire to tailor the rate and nature of the reports generated by a 2-D sensor. For example, hosts that have a low bandwidth connection to an RMI device may wish to restrict the conditions under which the device generates reports, so as to reduce reporting bandwidth requirements. The reporting mode is a 3-bit field that describes the various reporting models:

ReportingMode = '000': Continuous, when finger present

The absolute data source interrupt is asserted for every reporting period during which one or more fingers are touching the 2-D sensor. A final interrupt is asserted after the last finger has been lifted.

ReportingMode = '001': Reduced reporting mode

In this mode, the absolute data source interrupt is asserted whenever a finger arrives or leaves. Fingers that are present but basically stationary do not generate additional interrupts unless their positions change significantly. In specific, for fingers already touching the pad, the interrupt is asserted whenever the change in finger position exceeds either DeltaXPosThreshold or DeltaYPosThreshold.

Note: The contents of the PalmDetectThreshold register are used in this mode.

ReportingMode = '010': Finger-state change reporting mode

In this mode, the absolute data source interrupt is asserted whenever a finger arrives or leaves. Changes in finger position while a finger is down do not generate interrupts, regardless of their magnitude.

Notes:

- The contents of the DeltaXPosThreshold and DeltaYPosThreshold registers are ignored in this mode.
- The contents of the PalmDetectThreshold register are ignored in this mode.

ReportingMode = '011': Finger-presence change reporting mode

In this mode, the absolute data source interrupt is asserted whenever finger presence changes, meaning from the state of having at least one finger to none or the other way around. Increasing or decreasing finger count while there is still a finger on pad does not generate interrupts. Changes in finger position while a finger is down do not generate interrupts either, regardless of their magnitude.

Notes:

- The contents of the DeltaXPosThreshold and DeltaYPosThreshold registers are ignored in this mode.
- The contents of the PalmDetectThreshold register are ignored in this mode.

All other *ReportingMode* values are reserved.

AbsPosFilt (F11_2D_Ctrl0, bit 3)

When set to '1', this control bit enables filtering of the reported absolute position. When set to '0', absolute position filtering is disabled.

RelPosFilt (F11_2D_Ctrl0, bit 4)

When set to '1', this control bit enables filtering of the reported relative position changes. When set to '0', relative position filtering is disabled.

RelBallistics (F11_2D_Ctrl0, bit 5)

When set to '1', this control bit enables ballistics processing for the relative finger motion on the 2-D sensor. When set to '0', ballistics processing is disabled.

Dribble (F11_2D_Ctrl0, bit 6)

When set to '1', this control bit enables dribbling. When set to '0', dribbling is disabled.

Note: This bit has no effect unless HasDribble (F11_2D_Query5, bit 4) reports as '1'.

6.3.2. F11_2D_Ctrl1: palm and finger control

PalmDetectThreshold (F11_2D_Ctrl1, bits 3:0)

Specifies the threshold at which a wide finger is considered a palm. A value of 0 inhibits palm detection. The sensor inhibits its position reporting interrupts whenever either Wx or Wy is greater than or equal to the *PalmDetectThreshold*. This feature is only available when Reporting Mode is set to '001'.

MotionSensitivity (F11_2D_Ctrl1, bits 5:4)

This field specifies the threshold an anchored finger must move before it is considered no longer anchored. The settings are:

- 00 – Low motion sensitivity
- 01 – Medium motion sensitivity
- 10 – High motion sensitivity
- 11 – Infinite motion sensitivity

This control field is only available when the *HasAnchoredFinger* query reports as '1'. The measurements related to these settings are product-specific. Also, the ranges for Low, Medium, and High sensitivity may overlap.

ManualTrackingEn (F11_2D_Ctrl1, bit 6)

This 1-bit field specifies what entity, host or sensor firmware, is in control of determining which finger is the tracked finger:

- If '1', the host selects which finger is the tracked finger.
- If this bit is '0', the firmware will choose which finger is the tracked one.

This control field is only available when the *HasAnchoredFinger* query reports as '1'.

ManuallyTrackedFinger (F11_2D_Ctrl1, bit 7)

This 1-bit field indicates which finger is being tracked:

- If '0', Finger #0 is being tracked.
- If '1', then Finger #1 is being tracked.

Finger #0 is the finger reported in the first set of finger data registers (F11_2D_Data1.0 through F11_2D_Data5.0) while Finger #1 is the finger reported in the second set of finger data (F11_2D_Data1.1 thru F11_2D_Data5.1).

This control field is only meaningful when *ManualTrackingEn* (F11_2D_Ctrl1, bit 6) is enabled. This control field is only available when *HasAnchoredFinger* reports as '1'.

6.3.3. F11_2D_Ctrl2, 3: distance threshold

DeltaXPositionThreshold (F11_2D_Ctrl2)

DeltaYPositionThreshold (F11_2D_Ctrl3)

In ReportingMode '001' (Reduced Reporting), 2-D position update interrupts are inhibited unless the finger moves more than a certain threshold distance along either axis. These registers are used to define the thresholds for each axis. A value of 0 for both registers means that any change in position will cause an interrupt, similar to reporting mode '000' (continuous, when finger present).

6.3.4. F11_2D_Ctrl4: velocity control

Velocity (F11_2D_Ctrl4)

When *RelBallistics* is set to '1', this register defines the velocity ballistic parameter applied to all relative motion events. If *RelBallistics* is set to '0', this register is ignored.

6.3.5. F11_2D_Ctrl5: acceleration control

Acceleration (F11_2D_Ctrl5)

When *RelBallistics* is set to '1', this register defines the acceleration ballistic parameter applied to all relative motion events. If *RelBallistics* is set to '0', this register is ignored.

6.3.6. F11_2D_Ctrl6 through 9: maximum X and Y position control

SensorMaxXPos (F11_2D_Ctrl6,7)

SensorMaxYPos (F11_2D_Ctrl8,9)

These 12-bit fields are used to define a sensor's maximum X and Y positions. X positions are reported in the range 1 through *SensorMaxXPos*-1; Y positions are reported in the range 1 through *SensorMaxYPos*-1. A reported X position of 0 or *SensorMaxXPos*, or a reported Y position of 0 or *SensorMaxYPos*, indicates that the position is outside of reportable range.

Note: Not all 2-D sensors are capable of sensing these extreme values. For example, if a sensor cannot recognize values outside its reportable range, then it will not report positions equal to 0 or *SensorMaxX/YPosition*.

The units for the sensor position are arbitrary. A host may choose to have the maximum dynamic range by setting both of these fields to their maximum value: 4095. Hosts implementing 2-D clear sensors over an LCD display may choose to set these registers so that reportable positions match the number of pixels on an axis. Other choices, for example, allow a host to get positions reported in physical distance units like millimeters.

Notes:

- The two *SensorMaxXPos* registers (*F11_2D_Ctrl6* and *F11_2D_Ctrl7*) are a coherent group: To change either register, both registers must be written.
- The two *SensorMaxYPos* registers (*F11_2D_Ctrl8* and *F11_2D_Ctrl9*) are a coherent group: To change either register, both registers must be written.

6.3.7. F11_2D_Ctrl10, 11: gesture control

Each of these control registers is present only if at least one of the gestures it controls is supported (for example, *F11_2D_Ctrl10* is only present if *F11_2D_Query7* is non-zero, and *F11_2D_Ctrl11* is only present if *F11_2D_Query8* is non-zero). For a definition of each of the gestures referenced below, see the documentation for the corresponding query register *F11_2D_Query1* (see section 6.2.2.1).

SingleTapIntEn (F11_2D_Ctrl10, bit 0)

Setting *SingleTapIntEn* to '1' enables a gesture interrupt in response to a single-tap gesture. A *single-tap* gesture is also sometimes referred to as a *tap* gesture.

TapAndHoldIntEn (F11_2D_Ctrl10, bit 1)

Setting *TapAndHoldIntEn* to '1' enables a gesture interrupt in response to a tap-and-hold gesture.

DoubleTapIntEn (F11_2D_Ctrl10, bit 2)

Setting *DoubleTapIntEn* to '1' enables a gesture interrupt in response to a double-tap gesture.

EarlyTapIntEn (F11_2D_Ctrl10, bit 3)

Setting *EarlyTapIntEn* to '1' enables a gesture interrupt in response to an early-tap gesture.

FlickIntEn (F11_2D_Ctrl10, bit 4)

Setting *FlickIntEn* to '1' enables a gesture interrupt in response to a flick gesture.

PressIntEn (F11_2D_Ctrl10, bit 5)

Setting *PressIntEn* to '1' enables a gesture interrupt in response to a press gesture.

PinchIntEn (F11_2D_Ctrl10, bit 6)

Setting *PinchIntEn* to '1' enables a gesture interrupt in response to a pinch gesture.

PalmDetIntEn (F11_2D_Ctrl11, bit 0)

Setting *PalmDetIntEn* to '1' enables a gesture interrupt in response to a palm detect gesture.

RotateIntEn (F11_2D_Ctrl11, bit 1)

Setting *RotateIntEn* to '1' enables a gesture interrupt in response to a rotate gesture.

TouchShapeIntEn (F11_2D_Ctrl11, bit 2)

Setting *TouchShapeIntEn* to '1' enables a gesture interrupt in response to a TouchShape touch or lift.

ScrollZoneIntEn (F11_2D_Ctrl11, bit 3)

Setting *ScrollZoneIntEn* to '1' enables a gesture interrupt in response to motion on a ScrollZone.

MultiFingerScrollIntEn (F11_2D_Ctrl11, bit 4)

Setting *MultiFingerScrollIntEn* to '1' enables a gesture interrupt in response to multi-finger scrolling.

Note: Gestures are always detected and reported in the gesture data registers, regardless of the setting of the corresponding interrupt enables in F11_2D_Ctrl10 and 11. The interrupt-enable registers only define which gestures will participate in generating ATTN; they do not actually mask reporting of the data bits.

6.3.8. F11_2D_Ctrl12.*: sensor mapping control

The size of this register block is defined by the *MaximumElectrodes* query register (see section 6.2.2.3).

SensorMap (F11_2D_Ctrl12.size, bits 6:0)

The lower 7 bits of each *SensorMap* register map a particular ASIC sensor electrode to a logical electrode in the 2-D X or Y axes. The most significant bit, *XYSel*, indicates whether the mapping is to the X axis or to the Y axis. The size of the sensor map is described by the *NumberOfElectrodes* query.

If the *Configurable* query bit is ‘1’, then the host can arbitrarily assign the sensor mapping. If the *Configurable* query bit is ‘0’, then the sensor mapping is fixed by the module design, and can’t be changed. In either case, the total number of *SensorMap* registers is defined by the *MaximumElectrodes* query register (see section 6.2.2.3).

The logical electrode mapping begins with X_0 and extends up to X_{N_x-1} . The Y-electrodes begin immediately after the X-electrodes and are similarly order from Y_0 and extend up to Y_{N_y-1} . Each register position in the *SensorMap* selects a particular physical electrode to map to the logical electrode. The number of X and Y electrodes can either be read directly from the query registers for the case when the *Configurable* query bit is ‘0’, or can be inferred by the configuration of the *XYSel* bits.

When *XYsel* is ‘0’, the mapping applies to the X-axis. When ‘1’, the mapping applies to the Y-axis. Because the logical mapping begins with the X-axis, the first N_x registers will have *XYSel* set to ‘0’. The N_x+1 register will have *XYSel* set to ‘1’ to mark the beginning of the Y-axis mapping. The next N_y registers will have *XYSel* set to ‘1’. If the host desires to configure a sensor mapping that is smaller than the actual number of *SensorMap* registers, it should mark the end of the table by setting *XYSel* to ‘0’ for the first unused sensor after the last Y sensor mapping. See the example below for a 4X-by-3Y 2-D mapping.

Name	7	6	5	4	3	2	1	0
F11_2D_Ctrl12.0	XYSel=0				SensorMap=S0			
F11_2D_Ctrl12.1	XYSel=0				SensorMap=S7			
F11_2D_Ctrl12.2	XYSel=0				SensorMap=S1			
F11_2D_Ctrl12.3	XYSel=0				SensorMap=S6			
F11_2D_Ctrl12.4	XYSel=1				SensorMap=S10			
F11_2D_Ctrl12.5	XYSel=1				SensorMap=S11			
F11_2D_Ctrl12.6	XYSel=1				SensorMap=S12			
F11_2D_Ctrl12.7	XYSel=0				SensorMap=n/a			
F11_2D_Ctrl12.8	XYSel=0				SensorMap=n/a			

Figure 35. Function \$11 sensor mapping example

A 2-D device with four X electrodes and three Y electrodes is mapped onto a *configurable* 2-D device that supports up to a maximum of 9 sensors. The *XYsel* of ‘0’ indicates that for this sensor, the X-axis is mapped to the four physical electrodes S0, S7, S1, and S6. Similarly, the *XYsel* of ‘1’ indicates that the Y-axis is mapped to the three physical electrodes S10, S11, and S12. The subsequent setting of *XYsel* back to ‘0’ in mapping F11_2D_Ctrl12.7 marks the end of the entries in the sensor mapping table.

6.3.9. F11_2D_Ctrl13.*: reserved for Synaptics use

These registers are reserved for Synaptics use.

6.3.10. F11_2D_Ctrl14: sensitivity adjustment

This register is only present if the *HasSensitivityAdjust* bit in F11_Query_1 reports as '1'.

SensitivityAdjustment (F11_2D_Ctrl14, bits 4:0)

This 5-bit signed field allows a host to alter the overall sensitivity of a 2-D sensor. As shipped, the default *SensitivityAdjustment* of 0 will correspond to the factory recommended overall sensitivity setting for the sensor. A host may choose to make the sensor more or less sensitive than the factory setting by changing the value in this register. A positive value in this register will make the sensor more sensitive than the factory defaults, and a negative value will make it less sensitive.

HysteresisAdjustment (F11_2D_Ctrl14, bits 7:5)

This 3-bit unsigned field makes it possible to change the Z hysteresis setting for a 2-D sensor. By default, the factory recommended setting is '0'. Changing this setting to a value higher than '0' will increase the hysteresis by two Z units per count such that the maximum setting of '7' corresponds to an increase in hysteresis of 14 Z units.



WARNING: Setting this register to extreme values may render the sensor incapable of detecting touch or release events.

6.3.11. F11_2D_Ctrl15: maximum tap time

This control register is present only if *HasGestures* reports as '1' in the per-sensor query register F11_2D_Query1 (see section 6.2.2.1) and *HasEarlyTap*, *HasSingleTap*, *HasDoubleTap*, or *HasTapAndHold* reports as '1' in the per-sensor query register F11_2D_Query7.

Maximum Tap Time (F11_2D_Ctrl15, bits 7:0)

Determines the maximum duration of a tap, in 10-millisecond units. Maximum intertap-time (between the two taps of a double-tap gesture or between the tap and the hold of a tap-and-hold gesture) is proportional to this value.

6.3.12. F11_2D_Ctrl16: minimum press time

This control register is present only if *HasGestures* reports as '1' in the per-sensor query register F11_2D_Query1 (see section 6.2.2.1) and *HasPress* reports as '1' in the per-sensor query register F11_2D_Query7.

Minimum Press Time (F11_2D_Ctrl16, bits 7:0)

The minimum duration required for stationary finger(s) to generate a press gesture, in 10-millisecond units.

6.3.13. F11_2D_Ctrl17: maximum tap distance

This control register is present only if *HasGestures* reports as '1' in the per-sensor query register F11_2D_Query1 (see section 6.2.2.1) and *HasEarlyTap*, *HasSingleTap*, *HasDoubleTap*, *HasTapAndHold*, or *HasPress* reports as '1' in the per-sensor query register F11_2D_Query7.

Maximum Tap Distance (F11_2D_Ctrl17, bits 7:0)

Determines the maximum finger movement allowed during a tap, in 0.1-millimeter units.

Maximum allowed inter-tap movement is proportional to these values, as is maximum movement allowed during a press.

6.3.14. F11_2D_Ctrl18: minimum flick distance

This control register is present only if *HasGestures* reports as ‘1’ in the per-sensor query register F11_2D_Query1 (see section 6.2.2.1) and *HasFlick* reports as ‘1’ in the per-sensor query register F11_2D_Query7.

Minimum Flick Distance (F11_2D_Ctrl18, bits 7:0)

Determines the minimum finger movement for a flick gesture, in 1 mm units.

6.3.15. F11_2D_Ctrl19: minimum flick speed

This control register is present only if *HasGestures* reports as ‘1’ in the per-sensor query register F11_2D_Query1 (see section 6.2.2.1) and *HasFlick* reports as ‘1’ in the per-sensor query register F11_2D_Query7.

Minimum Flick Speed (F11_2D_Ctrl19, bits 7:0)

Determines the minimum finger speed for a flick gesture, in 10-millimeter/second units. Small values indicate lower speeds, large values indicate higher speeds.

6.4. Function \$11: data registers

Function \$11's data registers are divided into three groups: one which contains finger status and absolute position data, one for relative motion data, and one for gesture data. Within each group, the registers are time-coherent (see section 2.6), but the registers in one group are not guaranteed to be coherent with those in either of the other two.

6.4.1. Data register layout

The exact register layout depends on the specific features, as well as the number of fingers supported by the sensor. The register map construction rules are applied as follows:

1. One or more F11_2D_Data0 registers is placed first. The size of this register block can be calculated from the decoded *NumberOfFingers* field in F11_2D_Query1 (see section 6.2.2.1):

$$\text{F11_2D_Data0_registerCount} = \text{ceil}(\text{decodedNumberOfFingers}/4)$$
2. If the *HasAbs* field of F11_2D_Query1 reports as '1', then the block of data registers that describes the absolute finger position for a single finger (F11_2D_Data1 through Data5) is placed next. This block of registers is replicated once for each of the fingers that the sensor supports, as described by the *NumberOfFingers* field in F11_2D_Query1 (see section 6.2.2.1).
3. If the *HasRel* field of F11_2D_Query1 reports as '1', then the pair of data registers that describes the relative finger motion for a single finger (F11_2D_Data6, 7) is placed next. This block of registers is replicated once for each of the fingers that the sensor supports, as described by the *NumberOfFingers* field in F11_2D_Query1 (see section 6.2.2.1).
4. If F11_2D_Query7 is non-zero, then F11_2D_Data8 is placed next.
5. If F11_2D_Query7 or F11_2D_Query 8 is non-zero, then F11_2D_Data9 is placed next.
6. If the *HasPinch* or *HasFlick* field of F11_2D_Query7 reports as '1', then F11_2D_Data10 is placed next.
7. If the *HasRotate* field of F11_2D_Query8 or the *HasFlick* field of F11_2D_Query7 reports as '1', then both the F11_2D_Data11 and F11_2D_Data12 registers are placed next.
8. If the *HasTouchShapes* field of F11_2D_Query8 reports as '1', then one or more F11_2D_Data13 registers is placed next. The size of this register block can be calculated from the *NumberOfTouchShapes* field in F11_2D_Query10:

$$\text{F11_2D_Data13_registerCount} = \text{ceil}((\text{NumberOfTouchShapes}+1)/8)$$

9. If the *HasScrollZones* field of F11_2D_Query8 reports as '1', then F11_2D_Data14 and F11_2D_Data15 are placed next. If the *IndividualScrollZones* field of F11_2D_Query8 reports as '1', then F11_2D_Data16 and F11_2D_Data17 are placed next.

A two-finger example with individual Scroll Zones and 12 TouchShapes is shown in Figure 36:

Name	7	6	5	4	3	2	1	0
F11_2D_Data0.0	—	—	—	—	FingerState1	FingerState0		
F11_2D_Data1.0				X position (11:4)				
F11_2D_Data2.0				Y position (11:4)				
F11_2D_Data3.0		Y position (3:0)				X position (3:0)		
F11_2D_Data4.0		Wy (3:0)				Wx (3:0)		
F11_2D_Data5.0				Z (7:0)				
F11_2D_Data1.1				X position (11:4)				
F11_2D_Data2.1				Y position (11:4)				
F11_2D_Data3.1		Y position (3:0)				X position (3:0)		
F11_2D_Data4.1		Wy (3:0)				Wx (3:0)		
F11_2D_Data5.1				Z (7:0)				
F11_2D_Data6.0				X Delta				
F11_2D_Data7.0				Y Delta				
F11_2D_Data6.1				X Delta				
F11_2D_Data7.1				Y Delta				
F11_2D_Data8	—	Pinch	Press	Flick	EarlyTap	DoubleTap	TapAndHold	SingleTap
F11_2D_Data9		Gesture Finger Count		MultiFinger Scroll	ScrollZone	Shape	Rotate	PalmDetect
F11_2D_Data10				Pinch Motion / X Flick Distance				
F11_2D_Data11				Rotate Motion / Y Flick Distance				
F11_2D_Data12				Finger Separation / Flick Time				
F11_2D_Data13.0	Shape7	Shape6	Shape5	Shape4	Shape3	Shape2	Shape1	Shape0
F11_2D_Data13.1	—	—	—	—	Shape11	Shape10	Shape9	Shape8
F11_2D_Data14				X Lower Scroll Motion / Horizontal MultiFinger Scroll				
F11_2D_Data15				Y Right Scroll Motion / Vertical MultiFinger Scroll				
F11_2D_Data16				X Upper Scroll Motion				
F11_2D_Data17				Y Left Scroll Motion				

Figure 36. Function \$11 data register example (two fingers, scroll zones, and 12 TouchShapes)

6.4.2. Finger reporting

For sensors that support more than one finger, the data registers associated with a finger are determined at the time that the finger lands on the sensor. The data register block assigned to reporting that finger remains constant until the finger leaves. For example, consider the following sequence of events:

1. No fingers are present on the sensor.
2. A single finger lands on the sensor. This finger is defined to be the *primary finger*, or Finger0. The primary finger is reported using the first available block of absolute data registers, for example, Data1.0 through Data5.0.
3. With the primary finger still present, a second finger lands on the sensor, Finger1. This second finger gets assigned to the next available block of absolute data registers, for example, Data1.1 through Data5.1.

4. If the original finger were to lift while the second finger were to remain pressed, the second finger would become the primary finger, but that finger would remain being reported in the data registers to which it had originally been assigned; in the example above, this means that the primary finger is now reported in the Data1.1 through Data5.1 register block.
5. If another finger were to land while the second finger remained pressed, the new finger would be reported in the first available block of absolute data registers.
6. If the sensor uses Relative mode, the F11_2D_Data6.* and Data7.* registers are used to track relative movement of the fingers.

6.4.3. F11_2D_Data0.*: finger status data

The size of this register block can be calculated from *NumberOfFingers* field in F11_2D_Query1 (see section 6.2.2.1). The number of F11_2D_Data0 registers can be calculated as:

$$\text{F11_2D_Data0_registerCount} = \text{ceil}(\text{NumberOfFingers}/4)$$

The F11_2D_Data0 registers encode a 2-bit status field for each finger supported by the sensor. The *FingerStateN* fields are assigned one per finger, starting from least-significant bit-pairs to most-significant bit-pairs. A Function \$11 sensor always contains at least one F11_2D_Data0 register. If a sensor supports more than four fingers, a second F11_2D_Data0 register will be present containing the status information for the remaining fingers.

FingerStateN

Each finger has two status bits to describe its state. The encoding of the bits is:

‘00’ – The finger is not present.

‘01’ – The finger is present and the positional information associated with it is accurate.

‘10’ – The finger is present, however the positional information associated with it may be inaccurate.

‘11’ – This encoding is reserved for product-specific usage. Consult the product-specific documentation for more details.

6.4.4. F11_2D_Data1, 2: X and Y position data (MSB)

These data registers report the most-significant bits of the absolute X and Y position data. A host that is interested in minimizing bus bandwidth can read these two registers and obtain a rough estimate of the finger position. To obtain a fully accurate position report, F11_2D_Data3 should also be read.

6.4.5. F11_2D_Data3: X and Y position data (LSB)

This register is only present if the *AbsDataSize* field of F11_2D_Query5 reports as ‘00’ (see section 6.2.2.4).

This data register contains the least-significant bits for both the X and Y absolute position information. In combination with F11_2D_Data1 and F11_2D_Data2, full 12-bit X and Y position reports can be constructed. See section 6.3.6 for information on how to arrange to have these 12-bit positions scaled into more useful units like ‘pixels’ or ‘mm’.

When no finger is present on the sensor, the X and Y positions report the last known valid finger position.

The fields in this register are defined as follows:

X3:0 (F11_2D_Data3, bits 3:0)

In conjunction with F11_2D_Data1, the combined 12-bit field reports the horizontal position of the finger on the pad, where the origin is on the left side of the pad.

Y3:0 (F11_2D_Data3, bits 7:4)

In conjunction with F11_2D_Data2, the combined 12-bit field reports the vertical position of the finger on the pad, where the origin is on the lowest side of the pad.

6.4.6. F11_2D_Data4: finger width (W) data

This register is only present if the AbsDataSize field of F11_2D_Query5 reports as '00' (see section 6.2.2.4).

Wx (F11_2D_Data4, bits 3:0)

Wy (F11_2D_Data4, bits 7:4)

These fields report the estimated finger width as an unsigned integer, where 0 represents an extremely narrow finger and 15 represents an extremely wide contact such as a palm laid flat on the sensor. The ratio of *Wx* and *Wy* provides an estimate of the finger contact aspect ratio.

If the *FingerState* field for that finger corresponds to '10', then the W values for all the fingers on the sensor will be the same.

6.4.7. F11_2D_Data5: finger contact (Z) data

This register is only present if the AbsDataSize field of F11_2D_Query5 reports as '00' (see section 6.2.2.4).

Z (F11_2D_Data5, bits 7:0)

This field reports the amount of finger contact or finger signal strength, which often serves as a rough estimate of finger pressure.

When *Z* = 0, the position can't be measured and the X and Y Position registers are left unchanged. By default *Z* is taken as 0 whenever the device's built-in algorithms determine that no finger is present.

6.4.8. F11_2D_Data6.* , 7.*: finger motion deltas

These registers are only present if the *HasRel* field of F11_2D_Query1 reports as '1' (see section 6.2.2.1). This block of registers is replicated once for each of the fingers that the sensor supports.

These registers report finger motion as signed, 8-bit values for each reported finger. The delta registers accumulate motion until the host reads the delta registers. The motion accumulators will clip to +127 or -128 if the delta motion exceeds the 8-bit range.



Important: The DeltaX and DeltaY registers *must* be read as a sequential pair. Reading these registers has the side effect of clearing them.

DeltaX (F11_2D_Data6, bits 7:0)

This byte reports the amount of horizontal finger motion during a reporting period, where a positive *DeltaX* represents rightward motion (toward increasing absolute X positions).

DeltaY (F11_2D_Data7, bits 7:0)

This byte reports the amount of vertical finger motion during a reporting period, where a positive *DeltaY* represents upward motion (toward increasing absolute Y positions).

6.4.9. F11_2D_Data8, 9: gesture data

F11_2D_Data8 is only present if the F11_2D_Query7 register is non-zero. F11_2D_Data9 is only present if either the F11_2D_Query7 or the F11_2D_Query8 registers are non-zero.

For a definition of each of the gestures referenced below, see the documentation for the corresponding query register F11_2D_Query1 (see section 6.2.2.1).

When a sensor recognizes a gesture, the appropriate gesture bit is set in one of these data registers. If the corresponding gesture interrupt enable bit is set (see section 6.3.7), an ATTN interrupt is generated. The ATTN interrupt due to a gesture event needs to be processed in a timely fashion by the host, especially if the host desires to correlate the gesture with a position. For example, if the host is interested in correlating a *single-tap* gesture with its corresponding absolute finger position, the host should read the position registers with minimal latency after detecting the tap. If the host is too slow to respond, a finger may arrive again at some other location, and the host will get a false indication of the position where the tap occurred.

The fields in these data registers are defined as follows:

SingleTap (F11_2D_Data8, bit 0)

If this bit is set, then a single-tap gesture has completed.
This bit is automatically cleared when it is read.

TapAndHold (F11_2D_Data8, bit 1)

If this bit is set, then a tap-and-hold gesture has completed.
This bit is automatically cleared when it is read.

DoubleTap (F11_2D_Data8, bit 2)

If this bit is set, then a double-tap gesture has completed.
This bit is automatically cleared when it is read.

EarlyTap (F11_2D_Data8, bit 3)

If this bit is set, then an early-tap gesture has been detected.
This bit is automatically cleared when it is read.

An early-tap gesture is identical to a single-tap gesture except that the gesture is complete as soon as the finger lifts. In contrast, a single-tap gesture is not completed until enough finger-up time has passed to be sure that the tap is not part of a tap-and-hold or double-tap gesture.

Flick (F11_2D_Data8, bit 4)

If this bit is set, then a flick gesture has completed. This bit is automatically cleared when it is read. The X/Y distance and time associated with the flick gesture are reported in F11_2D_Data10 through F11_2D_Data12.

Press (F11_2D_Data8, bit 5)

If this bit is set, then a press gesture is active.

Note: This bit is not automatically cleared when it is read; it will remain set to 1 until the finger lifts or moves more than the maximum allowed press distance.

Pinch (F11_2D_Data8, bit 6)

If this bit is set, then a pinch gesture is either in progress or has completed.

This bit is cleared when it is read, but it will be set again if the pinch gesture continues.

The change in distance between the two fingers is reported in F11_2D_Data10.

PalmDetect (F11_2D_Data9, bit 0)

If this bit is set, then the palm detection gesture is active. A *palm* is defined to be any ‘large’ conductive object touching the 2-D sensor. An object is considered *large* if its width in either the X or Y axis that exceeds the *PalmDetectThreshold* (F11_2D_Ctrl1 bits 3:0).

Notes:

- This bit is *not* automatically cleared when it is read; it will remain set to 1 until the palm lifts from the sensor.
- Most data reporting is inhibited if a palm is detected. This gesture notifies the host when this condition persists.

Rotate (F11_2D_Data9, bit 1)

If this bit is set, then a rotate gesture is either in progress or has completed.

This bit is cleared when it is read, but it will be set again if the rotate gesture continues.

The accumulated finger motion distance is reported in F11_2D_Data11 and instantaneous relative finger separation is reported in F11_2D_Data12.

Shape (F11_2D_Data9, bit 2)

If this bit is set, then a finger has either touched or lifted from a TouchShape. The touched/lifted status of each TouchShape is reported in the F11_2D_Data13.* registers.

Note: This bit is automatically cleared when it is read, but the F11_2D_Data13.* registers are unaffected by reading this bit.

ScrollZone (F11_2D_Data9, bit 3)

If this bit is set, then finger motion in a ScrollZone is either in progress or has completed.

This bit is cleared when it is read, but it will be set again if the motion continues.

The accumulated finger motion distance is reported in F11_2D_Data14 through F11_2D_Data17.

MultiFingerScroll (F11_2D_Data9, bit 4)

If this bit is set, then a multi-finger scroll gesture is either in progress or complete. The accumulated finger motion distance is reported in F11_2D_Data14 and F11_2D_Data15.

Check the *GestureFingerCount* (F11_2D_Data9, bits 7:5) field to determine how many fingers were present when this gesture was detected.

GestureFingerCount (F11_2D_Data9, bits 7:5)

This field reports the actual number of fingers used in the reported gesture.

Table 3. Number of fingers used in the reported gesture

Encoding	Number of Fingers Reported
0	1
1	2
2	3
3	4
4	5
5	6
6	7
7	Reserved

For example, a two-finger press gesture would set the *Press* bit to '1', and also set the *GestureFingerCount* field to '1' (indicating two fingers present). A single-finger flick would set the *Flick* bit to '1', and also set the *GestureFingerCount* to '0' (indicating one finger).

6.4.10. F11_2D_Data10: pinch motion and X flick distance

F11_2D_Data10 is shared by pinch and flick gestures. This register is present if *HasPinch* field or the *HasFlick* field of F11_2D_Query7 reports as '1':

- When the *Pinch* bit of F11_2D_Data8 is set to '1', this register reports the change in distance between the two fingers since this register was last read. Negative values are reported when the fingers are moving closer together; positive values are reported when the fingers are moving apart. Units are in millimeters.
- When the *Flick* bit of F11_2D_Data8 is set to '1', this register reports the distance of flick gesture in X direction. Negative values are reported when the finger moves in negative X direction; positive values are reported when the finger moves in positive X direction. Units are in millimeters.

Note: This register is cleared to '0' when read.

6.4.11. F11_2D_Data11: rotate motion and Y flick distance

F11_2D_Data11 is shared by the rotate and flick gestures. This register is present if the *HasFlick* field of F11_2D_Query7 or the *HasRotate* field of F11_2D_Query8 reports as '1'.

- When the *Flick* bit of F11_2D_Data8 is set to '1', this register reports the distance of the flick gesture in Y direction. Negative values are reported when the finger moves in negative Y direction; positive values are reported when the finger moves in positive Y direction.
- When the *Rotate* bit of F11_2D_Data9 is set to '1', this register reports the accumulated distance of the rotate motion. The register is a signed quantity; clockwise motion is positive and counterclockwise motion is negative.

Notes:

- This register is cleared to '0' when read.
- Units are in millimeters.

6.4.12. F11_2D_Data12: finger separation and flick time

This register is only present if the *HasFlick* field of F11_2D_Query7 reports as '1' or the *HasRotate* field of F11_2D_Query8 reports as '1'. When the *Flick* bit of F11_2D_Data8 is set to '1', this register reports the total time of the flick gesture. Flick speed can be calculated in conjunction with X/ Y flick distance. Units are in 10-millisecond increments for flick and in millimeters for rotate.

Note: This register is cleared to '0' when read.

6.4.13. F11_2D_Data13.*: TouchShape status

These registers are only present if the *HasTouchShapes* field of F11_2D_Query8 reports as '1'. Each bit of these registers reports the state of a TouchShape: 0 = not touched, 1 = touched.

6.4.14. F11_2D_Data14: x lower scroll motion / MultiFinger horizontal scroll

This register is only present if the *HasScrollZones* field or the *HasMultiFingerScroll* field of F11_2D_Query8 reports as '1'. When the *ScrollZone* bit of F11_2D_Data9 is set to '1', this register reports the distance of the ScrollZone motion in the X direction. Negative values are reported when the finger moves in negative X direction; positive values are reported when the finger moves in positive X direction. When the *MultiFingerScroll* bit of F11_2D_Data9 is set to '1', this register reports the distance of the MultiFinger Scroll motion in the X (horizontal) direction.

Notes:

- This register is cleared to '0' when read.
- If the *IndividualScrollZones* field of F11_2D_Query8 reports as '0', this register reports the combined motion on the "X Lower" and "X Upper" ScrollZones. If the *IndividualScrollZones* field of F11_2D_Query8 reports as '1', this register reports only the motion on the "X Lower" ScrollZone.

6.4.15. F11_2D_Data15: y right scroll motion / MultiFinger vertical scroll

This register is only present if the *HasScrollZones* field or the *HasMultiFingerScroll* field of F11_2D_Query8 reports as '1'. When the *ScrollZone* bit of F11_2D_Data9 is set to 1, this register reports the distance of the ScrollZone motion in the Y direction. Negative values are reported when the finger moves in negative Y direction; positive values are reported when the finger moves in positive Y direction. When the *MultiFingerScroll* bit of F11_2D_Data9 is set to '1', this register reports the distance of the MultiFinger Scroll motion in the Y (vertical) direction.

Notes:

- This register is cleared to '0' when read.
- If the *IndividualScrollZones* field of F11_2D_Query8 reports as '0', this register reports the combined motion on the "Y Right" and "Y Left" ScrollZones. If the *IndividualScrollZones* field of F11_2D_Query8 reports as '1', this register reports only the motion on the "Y Right" ScrollZone.

6.4.16. F11_2D_Data16: x upper scroll motion

This register is only present if the *HasScrollZones* field of F11_2D_Query8 reports as '1' and the *IndividualScrollZones* field of F11_2D_Query8 reports as '1'. When the *ScrollZone* bit of F11_2D_Data8 is set to '1', this register reports the distance of the motion on the "X Upper" ScrollZone. Negative values are reported when the finger moves in negative X direction; positive values are reported when the finger moves in positive X direction.

Note: This register is cleared to '0' when read.

6.4.17. F11_2D_Data17: y left scroll motion

This register is only present if the *HasScrollZones* field of F11_2D_Query8 reports as '1' and the *IndividualScrollZones* field of F11_2D_Query8 reports as '1'. When the *ScrollZone* bit of F11_2D_Data8 is set to '1', this register reports the distance of the motion on the "Y Left" ScrollZone. Negative values are reported when the finger moves in negative X direction; positive values are reported when the finger moves in positive Y direction.

Note: This register is cleared to '0' when read.

6.4.18. Function \$11: interrupt source

Function \$11 defines an interrupt source for each data source that it contains. For example, an RMI device that contains a Function \$11 absolute 2-D data source and a gesture source, but no 2-D relative data source, will contain an interrupt source for the absolute and gesture data sources, but not for the relative data source.

The absolute data source of a 2-D sensor asserts an ATTN interrupt request on every report period where the interrupt conditions are satisfied by the current *ReportingMode* setting in control register F11_2D_Ctrl0 (see section 6.3.1). The reporting modes are defined to span a range of application requirements from very low reporting rates to very high reporting rates.

The relative data source of a 2-D sensor asserts an ATTN interrupt request on every report period that adds a non-zero amount of motion to either the X or Y delta motion accumulator registers.

Note: Because interrupt request bits are 'sticky', an interrupt request will remain at '1' even if a later backward finger motion subtracts the deltas down to zero again by the time the host reads the deltas; therefore, the host should be prepared occasionally to see (\$00, \$00) deltas even when an interrupt request is reported.

The gesture data source of a 2-D sensor asserts an interrupt request during every report period when a gesture has been either been detected (as in the case of pinch, press, or palm detect), or has been completed (as in the case of early-tap, single-tap, tap-and-hold, double-tap, and flick). The gesture source shares an interrupt request bit and an interrupt enable bit with the absolute data source.

Note: To clear interrupts caused by the gesture data source, the gesture flags registers F11_2D_Data8 and F11_2D_Data9 must be read in addition to reading the device Interrupt Status register.

6.5. Function \$11: command registers

Function \$11 has the following command register defined:

Name	7	6	5	4	3	2	1	0
F11_2D_Cmd0	—	—	—	—	—	—	—	ReZero

Figure 37. Function \$11 command register

ReZero (F11_2D_Cmd0, bit0)

Writing a ‘1’ to this field causes all 2-D sensors to revert to their “not touched” state. The host should never need to issue a Rezero command under normal conditions, because Synaptics devices handle zeroing completely automatically.

7. Function \$19: 0-D capacitive buttons

Function \$19 implements a 0-D sensing function, typically known as a capacitive button.

7.1. Function \$19: query registers

Query registers 0 and 1 contains general query information regarding Function \$19, button sensing.

Name	7	6	5	4	3	2	1	0
F19_Btn_Query0	—	—	—	—	—	Has Hysteresis Threshold	Has Sensitivity Adjust	Configurable
F19_Btn_Query1	—	—	—	ButtonCount				

Figure 38. Function \$19 query registers

7.1.1. F19_Btn_Query0: Configurable button query

The bits in register F19_Btn_Query0 are defined as follows:

Configurable (F19_Btn_Query0, bit 0)

If this field is '0', the button sensitivity control registers and sensor mapping control registers are not configurable at run-time:

- The *ButtonCount* field represents the exact number of buttons that exist in the product.

The sensor map control registers are read-only.

If this field is '1', the button enables and sensor mappings are configurable at run-time:

- The *ButtonCount* field represents the maximum number of buttons that can be enabled at once. Button numbers will always be in the range [0 through (*ButtonCount*-1)].
- The sensor map control registers are read-write.

HasSensitivityAdjust (F19_Btn_Query0, bit 1)

If this field is '1', the all-button sensitivity adjustment register F19_Btn_Ctrl5 exists.

HasHysteresisThreshold (F19_Btn_Query0, bit 2)

If this field is '1', the hysteresis adjustment register F19_Btn_Ctrl6 exists.

7.1.2. F19_Btn_Query1: ButtonCount query

The bits in register F19_Btn_Query1 are defined as follows:

ButtonCount (F19_Btn_Query1, bits 4:0)

For configurable products (*Configurable* reports as '1'), the *ButtonCount* represents the maximum number of buttons that can be enabled at one time. For non-configurable products (*Configurable* reports as '0'), the *ButtonCount* represents the exact number of buttons that are supported in the product.

7.2. Function \$19: control registers

These registers control the operation of the capacitive buttons.

Name	7	6	5	4	3	2	1	0
F19_Btn_Ctrl0	—	—	—	—	FilterMode		ButtonUsage	
F19_Btn_Ctrl1.0	—	—	—	—	IntEnBtn3	IntEnBtn2	IntEnBtn1	IntEnBtn0
F19_Btn_Ctrl2.0	—	—	—	—	SingleBtn3	SingleBtn2	SingleBtn1	SingleBtn0
F19_Btn_Ctrl3.0	—	SensorMap_Btn0						
F19_Btn_Ctrl3.1	—	SensorMap_Btn1						
F19_Btn_Ctrl3.2	—	SensorMap_Btn2						
F19_Btn_Ctrl3.3	—	SensorMap_Btn3						
F19_Btn_Ctrl4.0	Reserved_Btn0							
F19_Btn_Ctrl4.1	Reserved_Btn1							
F19_Btn_Ctrl4.2	Reserved_Btn2							
F19_Btn_Ctrl4.3	Reserved_Btn3							
F19_Btn_Ctrl5	—	—	—	Sensitivity Adjustment				
F19_Btn_Ctrl6	—	—	—	—	Hysteresis Threshold			

Figure 39. Function \$19 control registers for an example of *ButtonCount*=4

The example above shows a view of the control registers that would be present if the device reported a *ButtonCount* of 4. F19_Btn_Ctrl1, F19_Btn_Ctrl2, F19_Btn_Ctrl3, and F19_Btn_Ctrl4 are all replicated registers. This means that the number of these registers varies, depending upon how many buttons are defined for this product. For example, every product contains at least one button interrupt enable control register. The button enable bits are mapped into the button enable registers such that bit 0 of the first button enable register controls the enabling of button0, bit 1 controls button 1, and so on, with 8 button controls per register.

A product with 11 buttons would have to have two button interrupt enable control registers, F19_Btn_Ctrl1.0 and F19_Btn_Ctrl1.1. Note that there is just one register each for F19_Btn_Ctrl5, *Sensitivity Adjustment*, and F19_Btn_Ctrl6, *Hysteresis Threshold*; these controls affect all buttons.

7.2.1. Calculating the number of control registers

The layout of the F\$19 Button control registers always follows the general form: one general control register, followed by some number of button enable control registers, followed by some number of single button participation registers, followed by some number of sensor map control registers, followed by a single InterferenceThreshold control register, and ending with a single diagnostic control register. The number of button interrupt enable control registers can be calculated from the *ButtonCount* query:

```
F19_ButtonInterruptEnableRegisterCount = trunc((ButtonCount + 7) / 8);
```

The number of single button participation control registers is always identical to the number of button enable control registers:

```
F19_ButtonParticipationRegisterCount = F19_ButtonEnableRegisterCount;
```

The number of sensor map control registers (F19_Btn_Ctrl3 through F19_Btn_Ctrl6 in the example above) is always identical to *ButtonCount*:

```
F19_SensorMapRegisterCount = ButtonCount
```

The number of *Reserved* control registers (F19_Btn_Ctrl6 through F19_Btn_Ctrl10 in the example above) is always identical to *ButtonCount*:

7.2.2. F19_Btn_Ctrl0: general control

The fields in this control register are defined as follows:

ButtonUsage (F19_Btn_Ctrl0, bits 1:0)

This 2-bit field provides guidance about how the capacitive buttons are expected to be used in typical operation of the device. The reset default is '00', *Unrestricted usage*. For devices with only one capacitive button, the *ButtonUsage* field is effectively ignored.

ButtonUsage = '00': Unrestricted usage.

This value indicates that the user may touch the buttons in any combination. If the user is touching multiple buttons, all of those buttons are reported.

ButtonUsage = '01': Reserved.

ButtonUsage = '10': Strongest button only

This 'single button reporting mode' indicates that the user is expected to touch only one capacitive button at a time. In this mode, only one of the buttons in the participation group can report as being pressed at any one time. If the finger is proximate to several buttons at the same time, the device will attempt to choose the button with the strongest finger signal.

ButtonUsage = '11': First button only.

This 'single button reporting mode' indicates that the user is expected to touch only one capacitive button at a time. In this mode, only one of the buttons in the participation group can report as being pressed at any one time. If the finger is proximate to several buttons at the same time, the device will attempt to choose the first button that was touched for as long as that button remains touched.

If the first button touched is lifted while other buttons are still touched, it is undefined which of the still-touched buttons will be reported next.

Button usages '10' and '11' are advisory in the sense that some devices might not fully implement the "strongest button" or "first button" rules. In such devices, button usages '10' and '11' may be treated the same. However, all Function \$19 devices must ensure that no more than one button data bit is reported as '1' at the same time whenever the Button Usage field is set to '10' or '11'.

FilterMode (F19_Btn_Ctrl0, bit 2)

Capacitive buttons are always filtered to reduce the effects of electrical noise. Depending on the specific noise environment for a product, different filters may provide improved usability tradeoffs.

FilterMode = '00': Standard Filter

The majority of button products should use the standard filter. This filter is designed to produce the best balance of button responsiveness versus noise rejection.

FilterMode = '01': High Noise Rejection Filter

For systems subject to unusually high levels of electrical noise, this filter setting implements higher levels of noise rejection, but at the cost of increasing the time it takes to register button presses and releases, and decreasing the ability to detect fast button tap events. It should only be used when the standard filter mode is not suitable.

FilterMode = '10', '11':

These filter modes are *reserved* for device-specific filtering choices. They may not be present in all RMI devices. Consult the product specification for more information.

7.2.3. F19_Btn_Ctrl1.*: button interrupt enable control

Every product contains at least one button interrupt enable control register. The exact number of these registers is product-specific. See section 7.2.1 for details on calculating the total number of control registers in a product that contains RMI F\$19 Buttons.

The buttons in a product are numbered starting at 0, and continuing up to (*ButtonCount*-1). The button enable bits are mapped into the button enable registers such that bit 0 of the first button enable register controls the enabling of button0, bit 1 controls button1, and so on. If there are more than 8 buttons, then bit 0 of the next sequential button enable register controls button8, and so on.

Assigning a '1' to a button interrupt enable bit means that if the corresponding bit in the button data register changes state, the F\$19 button interrupt source (see section 7.3.2) generates an ATTN interrupt. Assigning a '0' to a button interrupt enable bit means that the corresponding bit in the button data register is ignored as part of the F\$19 ATTN interrupt generation process. This mechanism allows a host to do things like put the system into a low power mode where all buttons are ignored except for a single button that is defined to generate a wake-up ATTN interrupt.

7.2.4. F19_Btn_Ctrl2.*: single button participation control

Button modes '10' (strongest button mode) and '11' (first button mode) are referred to as *single button reporting modes* in that only one button is reported from a set of buttons that might be pressed at the same time. By default, the single button reporting modes select one button from among every possible button on the device. Certain products may find it advantageous to exclude certain buttons when calculating the result of a single button reporting mode. For example, a device might support buttons that allow a user to select from a class of related, but mutually exclusive operations like 'fast forward', 'rewind', 'pause', and 'play'. It would be natural to allow a user to select only one of these buttons at a time.

However, the same device might also contain a 'mute' or 'wireless' button. From a human interface point of view, a user should be allowed to press the 'wireless' or 'mute' buttons at the same time as they are holding down the 'fast forward' button. In that case, it would be desirable to exclude the 'wireless' and 'mute' buttons from the set of buttons participating in the single button reporting modes.

The button participation control register allows a host to select which buttons are excluded from consideration in the single button reporting selection process. Assigning a '1' to a button bit in this register means that the corresponding button is excluded from participating in the single button reporting modes, so the true state of the button will always be reported in the corresponding bit in a button data register. Assigning a '0' to a button bit in this register indicates that button reports for that bit are subject to the outcome of the single button reporting modes.

Every product contains at least one button participation control register. The number of registers and the numbering of the bits within the registers is always identical to the button enable control registers.

7.2.5. F19_Btn_Ctrl3.*: sensor map control

Every product contains at least one sensor map control register. The exact number of these registers is product-specific. See section 7.2.1 for details on calculating the total number of control registers in a product that contains RMI F\$19 Buttons.

The buttons in a product are numbered starting at 0, and continuing up to (*ButtonCount*-1). The number of sensor map control registers is always identical to *ButtonCount*. The set of sensor map registers is laid out such that the first sensor map control register applies to button0, the next sensor map control register applies to button1, and so on, up to the final SensorMap register which applies to button (*ButtonCount*-1).

Each of the sensor map control registers conforms to the following layout:

SensorMap (bits 4:0)

This 5 bit field maps a particular ASIC sensor electrode with a particular capacitive button. Writing some value N to this field has the effect of mapping ASIC sensor N to the button associated with the specific control register. The numbering of the sensor electrodes is based on the particular ASIC package. It is an undefined operation to assign a button to an ASIC sensor that does not exist in a given package.

If the *Configurable* field is '1', it indicates that the host can arbitrarily assign any capacitive sensor to a button by writing the sensor number into this field. See the ASIC documentation to find out the mapping of sensor numbers to ASIC pins.

If the *Configurable* field is '0' on a particular product, it means that the sensor mapping to ASIC pins is fixed by the module design, and can't be changed. As a result, these mapping registers are Read-Only. In that case, the registers will still report what sensor is assigned to each capacitive button on that product, but the host will be unable to make changes to those mappings.

Regardless of the state of the *Configurable* field, the default values for these sensor mapping registers are always product-specific; consult the product's documentation for more details.

The register map below assumes a sample product that is defined to have 4 buttons that can select from 28 ASIC sensors, S0 through S27:

Name	7	6	5	4	3	2	1	0
F19_Btn_Ctrl0	—	—	—	—	—	—	ButtonUsage='10'	
F19_Btn_Ctrl1.*	—	—	—	—	IntEnBtn3=0	IntEnBtn2=1	IntEnBtn1=1	IntEnBtn0=1
F19_Btn_Ctrl2.*	—	—	—	—	SingleBtn3=0	SingleBtn2=0	SingleBtn1=1	SingleBtn0=1
F19_Btn_Ctrl3.0	—	SensorMap_Btn0 = 15						
F19_Btn_Ctrl3.1	—	SensorMap_Btn1 = 27						
F19_Btn_Ctrl3.2	—	SensorMap_Btn2 = 0						
F19_Btn_Ctrl3.3	—	SensorMap_Btn3 = 6						

Figure 40. Function \$19 sensor map control registers for an example of ButtonCount=4

In the example shown in Figure 40, the first sensor map control register at F19_Btn_Ctrl3 corresponds to button0. The *SensorMap* field in that register contains the value 15, meaning that button0 is associated with ASIC sensor pin S15.

In the same fashion, button1 is associated with ASIC sensor pin S27, button2 is associated with ASIC sensor pin S0, and button3 is associated with ASIC sensor S6. The sample register map also indicates that the interrupt for button3 is '0' (disabled), so events on button3 will not generate interrupts, although the state of Button3 will still be reported in the data register. For disabled buttons, the *SensorMap* value in the corresponding sensor map control register is unimportant. The *ButtonUsage* is '10', or the strongest button mode. Register F19_Btn_Ctrl2 indicates that button0 and button1 are participating in the single button modes. If a user simultaneously touches all four buttons on this device:

- Either button0 or button1 is reported as being pressed, depending on which has the strongest signal.
- Button2 reports as being pressed because it is not part of the single button participation group.
- Button3 still reports as being pressed, even though its corresponding interrupt bit is not enabled.

7.2.6. F19_Btn_Ctrl4.*: reserved

These registers are reserved for Synaptics use. Do not alter the contents of these registers when updating the device flash configuration.

7.2.7. F19_Btn_Ctrl5: all-button sensitivity adjustment

The fields in this control register are defined as follows:

SensitivityAdjustment (F19_Btn_Ctrl5, bits 4:0)

This 5-bit field provides a global sensitivity adjustment to the contacting and releasing thresholds applicable to all buttons. The value is signed, from -16 to 15, with -16 the least sensitive threshold for all buttons and +15 the most sensitive threshold for all buttons. The default adjustment is 0.

7.2.8. F19_Btn_Ctrl6: all-button hysteresis threshold

The fields in this control register are defined as follows:

HysteresisThreshold (F19_Btn_Ctrl6, bits 3:0)

This 4-bit field introduces hysteresis to button reporting when operating in the *Strongest Button Mode*. In that ‘single button reporting mode’, only the button with the strongest finger signal is reported at any one time.

By applying *Hysteresis Threshold*, a new strongest button is selected only if it is stronger by the threshold amount than the currently reported one (or, the strength of the currently reported one falls to be less than another button by the threshold amount).

This mechanism avoids fast alternative reporting among multiple fingers when they have a close strength of signal. *HysteresisThreshold* ranges from 0 to 15, where 0 means disabled. The default threshold is 0.

7.3. Function \$19: data registers

Function \$19 always has one data source.

Each bit in the capacitive button data registers is ‘1’ if the button is being touched, or ‘0’ if the button is not being touched.

7.3.1. Calculating the number of data registers

The number of button-reporting data registers depends on the *ButtonCount* query:

$$F19_DataRegisterCount = \text{trunc}((\text{ButtonCount} + 7) / 8)$$

The data registers of a capacitive button sensor are shown in Figure 41, with the example of a *ButtonCount* of 12.

Name	7	6	5	4	3	2	1	0
F19_Btn_Data0.0	Btn7	Btn6	Btn5	Btn4	Btn3	Btn2	Btn1	Btn0
F19_Btn_Data0.1	—	—	—	—	Btn11	Btn10	Btn9	Btn8

Figure 41. Function \$19 capacitive button data registers (for example of 12 buttons)

7.3.2. Function \$19: interrupt source

Function \$19 implements one interrupt source. Function \$19 asserts an interrupt request whenever any button bit in any of its button data registers changes from a ‘0’ to a ‘1’ or from a ‘1’ to a ‘0’, and the corresponding button interrupt enable bit is a ‘1’. Because an interrupt request is a “sticky” bit, interrupt requests read as ‘1’ if any button bit has changed since the last time the data registers were read, even if the button bit has changed back to its previous value since that time. An interrupt request is asserted synchronously with the report rate of other capacitive sensors in the device.

7.4. Function \$19: command registers

Function \$19 implements a single command register.

Name	7	6	5	4	3	2	1	0
F19_RMI_Cmd0	—	—	—	—	—	—	—	Rezero

Figure 42. Function \$19 command register

Rezero Command (F19_Btn_Cmd_0, bit 0)

Writing a ‘1’ to this field causes all button sensors to revert to their “not touched” state. The host should never need to issue a Rezero command under normal conditions, because Synaptics devices handle zeroing completely automatically.

8. Function \$30: LED/GPIO Control

Function \$30 implements a group of general purpose input/output pins, as might be used for input buttons, LED control and so on. Each pin is configured as either an input/output (GPIO) or an adjustable-current output (LED). The adjustable-current LED pins can be programmed for a variety of waveforms. Although an adjustable-current output is referred to as an LED, there is nothing requiring the controlled component to be a light-emitting diode; in fact, simple LEDs that do not require accurate or current-limited brightness control may be implemented by the host as a GPIO output.

The same pin can be configured and controlled as either a GPIO or LED, which is the flexibility needed for a general-purpose input-output device. A custom product typically defaults each pin as an LED, a GPI, or a GPO depending on the pin's intended purpose, but the flexibility to change it remains.

8.1. Function \$30: power management

Function \$30 interacts with the power management features described in section 3.2.1. The device does not doze when any LED is active or when any LED is ramping down.

8.2. Function \$30: query registers

Query registers 0 and 1 contain general query information regarding Function \$30, GPIO/LED.

Name	7	6	5	4	3	2	1	0
F30_GPIO_Query0	—	—	HasGpio DriverControl	HasHaptic	HasGpio	HasLed	HasMappable Buttons	—
F30_GPIO_Query1	—	—	—	GpioLedCount				

Figure 43. Function \$30 GPIO/LED query registers

HasMappableButtons (F30_GPIO_Query0, bit 1)

The *HasMappableButtons* bit is '0' if a GPIO/LED can't be controlled by capacitive buttons. If the *HasMappableButtons* bit is '1' then the each GPIO/LED pin may be controlled by a capacitive button. The programming of which button controls a GPIO/LED is described in section 8.3.8. This bit affects the number of such registers: there is either zero total or one register per GPIO/LED.

HasLed (F30_GPIO_Query0, bit 2)

This bit is set when the function support basic LED operations. When this bit is '0', the registers associated with LED operation are not present in the register map.

HasGpio (F30_GPIO_Query0, bit 3)

This bit is set when the function support basic GPI and GPO operations. When this bit is '0', the registers associated with GPI and GPO operation are not present in the register map.

HasHaptic (F30_GPIO_Query0, bit 4)

This bit is set when the function supports haptic operations. If the *HasHaptic* bit is '1' then any capacitive button may trigger a timed on-off change of the GPIO/LED pin. The *HasHaptic* bit is '0' if the haptic timed output pulse option is not available.

HasGpioDriverControl (F30_GPIO_Query0, bit 5)

This bit is set when the function supports programming GPIO pin/driver characteristics. See section 8.3.7 for information on the GPIO/LED control registers that define pin programming.

GpioLedCount (F30_GPIO_Query1, bits 4:0)

The GPIO/LED Count reports the number of GPIO/LED pins available. Several types of Function \$30 registers use this value to calculate their register space size.

8.3. Function \$30: control registers

These registers control the operation of the GPIO/LED pins. A ‘.’ indicates a variable-sized register block. Every product contains at least one of a variable-sized register block. The bits are mapped into the registers such that, for example, bit 0 of the GPIO/LED Select register controls the enabling of pin0, bit 1 controls pin 1, and so on. See section 8.3.1 for a description of how to determine the exact number of control registers for a product. See section 8.6 for an example of a register layout using 11 GPIO/LED pins.

8.3.1. Calculating the number of control registers

The GPIO/LED function has a large number of registers. Some of the control register areas have one bit per GPIO/LED (such as the F30 GPIO/LED Select registers) and some have an entire register per GPIO/LED.

For the registers that have one bit per pin, the number of registers is calculated from the GpioLedCount Query as follows

```
F30_???_RegisterCount = trunc((GpioLedCount + 7) / 8);
```

Where there is one register per GPIO/LED, the number of registers is

```
F30_???_RegisterCount = GpioLedCount;
```

8.3.2. F30_GPIO_Ctrl0.*: GPIO/LED select

Every Function \$30 usually contains at least one GPIO/LED Select register; this register is only present when both *HasLed* and *HasGpio* query bits are ‘1’. The ‘.’ indicates a variable-sized register block. See section 8.3.1 for a description of how to determine the exact number of control registers for a product.

The bits are mapped into the registers such that, for example, bit 0 of the GPIO/LED Select register controls the enabling of pin0, bit 1 controls pin 1, and so on. The reset default is ‘0’.

Name	7	6	5	4	3	2	1	0
F30_GPIO_Ctrl0.*	—	—	—	—	—	—	—	LedSel0

Figure 44. Function \$30 pin select register

Each GPIO/LED pin is configurable at run time as either a GPIO or a LED. For each bit, if bit = 0 then the associated pin is GPIO; if bit = 1 then the associated pin is LED.



Caution: This register should typically be written before any other register. Otherwise there may be unexpected behavior. If a pin’s type is changed by writing this register then any control registers associated with the pin are set to their default values; this means both the associated GPIO and LED register values are reset. For example, if a pin is changed from a LED to a GPIO then the pin’s GPIO control registers will go to their default state (high impedance) and the pins LED registers such as pattern and brightness will go to their default state. This is because the GPIO and LED functionality shares resources. This register is almost never changed after it is initialized; this behavior is not a limitation.

8.3.3. F30_GPIO_Ctrl1: GPIO/LED general control

Every Function \$30 contains one GPIO/LED General Control register. This register reset default is ‘0’, and can be left at 0 in most applications.

Name	7	6	5	4	3	2	1	0
F30_GPIO_Ctrl1	—	—	Halted	Halt	—	—	—	GpiDebounce

Figure 45. Function \$30 General Control register

GpiDebounce (F30_GPIO_Ctrl1, bit 0)

If this bit is set to ‘1’, the device applies a debouncing algorithm to all the GPI inputs. The exact debouncing algorithm is device-specific, but in general debouncing strives to eliminate brief glitches in the input signal due, for example, to “bounce” in the contacts of a mechanical switch. If *GpiDebounce* is not specified, there is still some debounce because transitions are only reported at the report rate.

Halt (F30_GPIO_Ctrl1, bit 4)

Writing this bit to a ‘1’ freezes this function’s operation. All LED ramping and animation are stopped. All GPIO input and output operations are stopped. If ramping and animation are frozen, each LED holds at whatever intensity it has at the moment the freeze bit is set, even if the LED is halfway through a ramp operation. When synchronized behavior is required, for example setting several LEDs active, this bit should be used. When this bit is written to a ‘0’, the ramps will continue where they left off. There is a delay after setting this bit until this function halts. The halted bit should be monitored for this status.

Halted (F30_GPIO_Ctrl1, bit 5)

This bit is a status indicator that acknowledges that this function is halted. Sometime after the *Halt* bit is set or reset, this bit will be set or reset.

8.3.4. F30_GPIO_Ctrl2.* and F30_GPIO_Ctrl3.*: GPIO input/output mode control

Every Function \$30 usually contains at least one GPIO Input/Output Mode register pair; these registers are only present if the *HasGpio* query bit is ‘1’. See section 8.3.1 for a description of how to determine the exact number of control registers for a product.

For each GPIO/LED there are two bits to specify the mode, *DirN* and *DataN*. These bits are split between registers as shown below. The reset default of *DirN* and *DataN* is typically ‘00’ (high-impedance).

Name	7	6	5	4	3	2	1	0
F30_GPIO_Ctrl2.0	—	—	—	—	—	—	—	Dir0
F30_GPIO_Ctrl3.0	—	—	—	—	—	—	—	Data0

Figure 46. Function \$30 GPIO input/output control register

These registers control the input/output modes of pins configured as GPIO. For a pin programmed as an LED, writes are ignored and reads return undefined values.

The Data N and Direction N bits together control the mode and output state of GPIO # N , as follows:

Table 4. Function \$30 GPIO control settings

Dir N	Data N	State of GPIO # N
0	0	High-impedance state (typically input mode)
0	1	Pull-up resistor (typically input mode)
1	0	Driving digital '0'
1	1	Driving digital '1'

Writes to this mode register for pins that are the target of a capacitive button-to-GPIO mapping are ignored as described in section 8.3.8.

To write to both Dir N and Data N without causing brief "glitches" on the output pins, perform the writes in this order:

1. When switching a pin's direction from input to output, write Data N followed by Dir N .
2. When switching a pin's direction from output to input, write Dir N followed by Data N .

Alternatively you can do the following:

1. Set the Halt bit (F30_GPIO_Ctrl1 bit 4) to 1.
2. Write the registers.
3. Clear the Halt bit to 0.

The state of an input is read through the GPI/LED data registers; see section 8.4 for more information.

8.3.5. F30_GPIO_Ctrl4.*: LED active control

Every Function \$30 usually contains at least one LED Active register; this register is only present if the *HasLed* query bit is '1'. The '.' indicates a variable-sized register block. See section 8.3.1 for a description of how to determine the exact number of control registers for a product.

The bits are mapped into the registers such that, for example, bit 0 of the LED Active register represents the setting for LED 0, bit 1 describes LED 1, and so on. See section 8.3.1 for information on determining the exact sizes. The reset default is '0'.

Name	7	6	5	4	3	2	1	0
F30_GPIO_Ctrl4.*	—	—	—	—	—	—	—	LedAct0

Figure 47. Function \$30 LED Active register

Reading this register returns which LEDs are active. The exact on/off behavior is specified in the LED control registers; see section 8.3.7 for more information. For pins programmed as a GPIO, writes are ignored and reads return undefined values. Writes are also ignored and reads are undefined if the LED is the target of a Button-to-GPIO mapping as described in section 8.3.8.

Writing the LED Active N bit to '1' sets LED# N to the Active state. The LED will turn on over a specified period of time or animate in a specified pattern. The amount of LED current corresponding to the "on" state for an LED is set by the per-LED Control registers. The data register indicates when a ramping operation is in progress; this applies to LEDs that turn on over a period of time.

Writing the LED Active N bit to '0' sets LED # N to the inactive state. The LED will turn off either immediately or over a specified period of time.

In RMI Function \$30, the “off” state is always represented by zero sink current on the LED pin. The data register indicates when a ramping operation is in progress; this applies to LEDs that turn off over a period of time.

8.3.6. F30_GPIO_Ctrl5.*: LED ramp period control

Every Function \$30 usually contains two or six Ramp Period registers, depending on the *ExtendedPatterns* query bit; this register is only present if the *HasLed* query bit is ‘1’. When the *ExtendedPatterns* bit is ‘0’, there are two registers, and when it is ‘1’, there are six registers. The reset default is ‘0’ for each register:

Name	7	6	5	4	3	2	1	0
F30_GPIO_Ctrl5.0	RampPeriodA							
F30_GPIO_Ctrl5.1	RampPeriodB							

Figure 48. Function \$30 LED Ramp Period registers

These registers specify the ramp rates as used by the LED patterns specified in the next section. Each unit of a period specifies 10 milliseconds, so the range of a ramp period is 0 to approximately 2.5 seconds. \$00 indicates an instantaneous transition and \$FF is approximately 2.5 seconds.

8.3.7. F30_GPIO_Ctrl6.*: GPIO/LED control

Each GPIO/LED usually has a register that programs something about the PIN operation. The ‘.*’ indicates that this is a variable-sized register; each LED is mapped to a register such that, for example, register 6.0 of the LED Active register block represents the settings for LED 0, register 6.1 describes LED 1, and so on. See section 8.3.1 for information on determining the exact sizes.

There are two situations when this register is preset:

- *HasLed* query bit is ‘1’, or
- *HasLed* is ‘0’ and *HasGpioDriverControl* is ‘1’.

The reset default is ‘0’ for each register. Each individual register in this register block controls either an LED or a GPIO, depending on whether the associated GPIO pin is programmed as an LED or a GPIO. This is typically selected with the GPIO/LED Select register; see section 8.3.2 for more information. The following subsections describe the two register layouts, one for a GPIO and one for LED.

8.3.7.1. F30_GPIO_Ctrl6.*: GPIO control

This is the register description for controlling a GPIO.

Name	7	6	5	4	3	2	1	0
F30_GPIO_Ctrl6.*	—	STRPU	—	STRPD	—	SPCTRL	—	—

Figure 49. Function \$30 per-GPIO Control register

STRPU (F30_GPIO_Ctrl6.*, bit 6)

This field defines the strong pull-up enable:

- If ‘0’, a weak resistive pull-up strength.
- If ‘1’, a strong resistive pull-up strength.

The weak pull-up strength consumes less power. This is only meaningful if the GPIO Input/Output Mode specifies a pull-up resistor.

STRPD (*F30_GPIO_Ctrl6.**, bit 4)

This field defines the strong pull-down strength:

- If ‘0’, there is no effect.
- If ‘1’, there is double the maximum uncontrolled GPIO sink current.

This bit is provided as an alternative to the controlled LED intensity control in situations where a current greater than 12 mA is desired and accuracy is not critical. The reason for the low accuracy is that the sink current is determined by the LED drop and series resistance of external circuit. Thus, in applications where the GPIO pull-down driver is used to sink current, the field that is a concatenation of STRPD and SPCTRL serves as a coarse control for the maximum uncontrolled sink current.

SPCTRL (*F30_GPIO_Ctrl6.**, bits 3:0)

This field defines the output driver strength control. This 3-bit field controls the drive speed (or drive strength) of the GPIO output driver. A higher value for the field corresponds to a higher drive strength.

8.3.7.2. F30_GPIO_Ctrl6.*: LED control

This is the register description for controlling an LED.

Name	7	6	5	4	3	2	1	0
F30_GPIO_Ctrl6.*	Pattern				Brightness			

Figure 50. Function \$30 per-LED Control register

These registers control the intensity and waveform/animation of the selected LED. There are two styles specified by the patterns: continuous animation while the LED is active and secondly waveforms that follow the LED active bit.

Brightness (*F30_GPIO_Ctrl6.**, bits 3:0)

The Brightness field indicates the target intensity level when the LED is enabled:

- \$0 represents fully off,
- \$1 - \$E represent intermediate intensities evenly or approximately-evenly spaced (in units of human-perceived brightness) between fully off and fully on, and
- \$F represents fully on.

Pattern (*F30_GPIO_Ctrl6.**, bits 7:4)

The Pattern field takes one of the following values:

Pattern = ‘0000’: Rise and fall period A.

In this setting, when the LED is activated by setting the LED Active *N* bit in the LED Active register, the LED ramps to the intensity indicated by the Brightness field over a time determined by *RampPeriodA* of the LED Ramp Period register and then holds at that intensity. When the LED is deactivated by clearing the LED Active *N* bit, the LED ramps down to the “off” state over a time determined by *RampPeriodA* and then remains “off.”

The ramp begins immediately after the write to the LED Active control registers or the Per-LED Control register, and may be unsynchronized (out of phase) with other ongoing ramps or animations. To ramp several LEDs synchronously, use the `LedHalt` bit of the GPIO/LED General Control register. The GPI/LED Data registers provide ramp busy status.

The LED's ramp busy bit shows '1' after the LED active bit is written, and stays at '1' until the LED brightness reaches either the target level or off, depending on the direction of the ramp.

Pattern = '0001': Rise and fall period B.

This pattern is like pattern '0000', except that the *RampPeriodB* parameter determines the ramp rate instead of *RampPeriodA*.

Pattern = '0010': Rise period A, fast fall.

In this setting, when the LED is activated the LED ramps to the intensity indicated by the *Brightness* field over a time determined by *RampPeriodA* and then holds at that intensity. When the LED is deactivated, the LED switches immediately to the "off" state.

Pattern = '0011': Rise period B, fast fall.

This pattern is like pattern '0010', except that the *RampPeriodB* parameter determines the ramp rate instead of *RampPeriodA*.

Pattern = '0100': Fast rise, fall period A.

In this setting, when the LED is activated the LED switches immediately to the intensity indicated by the *Brightness* field and then holds at that intensity. When the LED is deactivated, the LED ramps down to the "off" state over a time determined by *RampPeriodA* and then remains "off."

Pattern = '0101': Fast rise, fall period B.

This pattern is like pattern '0100', except that the *RampPeriodB* parameter determines the ramp rate instead of *RampPeriodA*.

Pattern = '0110': Ramping animation.

In this setting, when the LED is activated the LED ramps to the intensity indicated by the *Brightness* field over a time determined by *RampPeriodA*, and then holds at that intensity for a time determined by *RampPeriodB*. The LED then ramps down to the "off" state over *RampPeriodA* and remains off for *RampPeriodB*. This cycle repeats continuously for as long as the LED is activated. When the LED is deactivated, the LED switches immediately to the "off" state.

Pattern = '0111': Pulsed animation.

In this setting, when the LED is activated the LED pulses between the "on" and "off" states. The LED switches immediately to the intensity indicated by the *Brightness* field, then remains at the target intensity for a time determined by *RampPeriodB*. Then the LED switches immediately to the "off" state and remains "off" for a time determined by *RampPeriodA*. This cycle repeats continuously for as long as the LED is activated. When the LED is deactivated, the LED switches immediately to "off" state.

8.3.8. F30_GPIO_Ctrl7.*: button-to-GPIO mapping and control

Each GPIO/LED has one register that programs a capacitive button to control the GPO/LED output, as described in section 6. If the *HasMappableButtons* query bit is ‘0’ then these Button-to-GPIO Mapping and Control registers are not present in the register space.

The ‘.*’ indicates a variable-sized register block. Every product contains at least one Button Mapping register. Each LED is mapped to a register such that, for example, register 7.0 of the Button Mapping register block represents the settings for LED 0, register 7.1 describes LED 1, and so on. See section 8.3.1 for information on determining the exact sizes. The reset default is ‘0’ for each register.

Name	7	6	5	4	3	2	1	0
F30_GPIO_Ctrl7.*	OpenDrain	Invert	Valid			CapacitiveButtonNumber		

Figure 51. Function \$30 per-GPIO/LEDButton Mapping register

The precise affect of the capacitive button on the GPIO/LED is defined by the *ButtonPolarity* and *OpenDrain* bits. For a mapping to an LED, the button state sets the corresponding LED Active register bit as described in section 8.3.5. For a mapping to a GPO the output follows the capacitive button.

CapacitiveButtonNumber (F30_GPIO_Ctrl7.*, bits 4:0)

This is the capacitive button number that affects the GPIO/LED pin that corresponds to this register. A value of 0x1F specifies that any capacitive button affects the GPIO/LED. The precise meaning is that all the capacitive buttons are ORed together into a single virtual button which is used to affect the corresponding GPIO/LED.

Valid (F30_GPIO_Ctrl7.*, bit 5)

When this bit is ‘1’, the output associated with this GPIO/LED is controlled by the specified button.

Invert (F30_GPIO_Ctrl7.*, bit 6)

When this bit is ‘1’, the state of the button (finger present or finger absent) is inverted before affecting the GPIO/LED.

OpenDrain (F30_GPIO_Ctrl7.*, bit 7)

For pins programmed as GPIO and controlled by a button, the pin’s mode (see section 8.3.4 for a description of the input and output modes) is affected by the button state; the button state overrides bits in that register. Set this bit to ‘1’ if the GPO is open-drain.

Implementation note: when this bit is ‘0’ the *DirN* bit is forced to ‘1’ and the button state overrides the mapped GPIO’s *DataN* bit. When this bit is a ‘1’ the *DataN* bit is forced to ‘0’ and the button state overrides the mapped GPIO’s *DirN* bit. Regardless of which of the two GPIO control bits is overridden, the resulting Data and Dir bits determine the behavior of the GPIO pin, as described in section 8.3.4.

Example 1: Active low “Totem Pole” button – *InvertN* = 1, *OpenDrainN* = 0

Example 2: SPST button connect to GND, - *InvertN* = 0, *OpenDrainN* = 1

Note: This bit is only used if the corresponding pin is programmed as a GPIO. It is not used if it is programmed as an LED.

Note: The same button may affect more than one GPIO/LED; for example, consider a tri-color LED. It is guaranteed that all the affected GPIO/LED animations/waveforms start simultaneously.

8.3.9. F30_GPIO_Ctrl8.*: haptic enable control

If the *HasHaptic* bit in Query Register 0 is set to 1, then Function \$30 contains at least one Haptic Enable register. See section 8.3.1 to determine the exact sizes.

Setting a bit in this register enables haptic output for the corresponding GPIO pin. If more than one bit is set, then the haptic pulse is output simultaneously to all enabled pins. The *CapacitiveButtonNumber* parameter for a GPIO enabled as a haptic output (see F30_GPIO_Ctrl7, bits 4:0) specifies which capacitive button to monitor. When multiple bits are set the corresponding *CapacitiveButtonNumbers* should be the same.

When any button state bit goes high, a one-shot timer is triggered/re-triggered. The timer state controls the state of a GPIO or LED, depending on *LedSel*. The *ButtonPolarity* and *OpenDrain* bits can be set for each GPIO pin.

Name	7	6	5	4	3	2	1	0
F30_GPIO_Ctrl8.0	HapticEn7	HapticEn6	HapticEn5	HapticEn4	HapticEn3	HapticEn2	HapticEn1	HapticEn0
F30_GPIO_Ctrl8.1						HapticEn10	HapticEn9	HapticEn8

Figure 52. Function \$30 Haptic Enable registers

8.3.10. F30_GPIO_Ctrl9: haptic duration control

If the *HasHaptic* bit in Query Register 0 is set to '1', then Function \$30 contains a HapticDuration register. The duration of the haptic output to the GPIO/LED pin is defined by the HapticDuration register.

Name	7	6	5	4	3	2	1	0
F30_GPIO_Ctrl9	HapticDuration							

Figure 53. Function \$30 Haptic Duration register

Each unit of duration represents 10 milliseconds, so the range of a haptic output pulse is 0 to approximately 2.5 seconds. \$00 disables the output, \$01 is a 10 millisecond output pulse and \$FF is approximately 2.5 seconds.

8.4. Function \$30: data registers

Function \$30 has a single data register type which is shared by GPI input data and LED ramping state information. The ‘.’ indicates a variable-sized register block. Every product contains at least one GPIO data register.

Name	7	6	5	4	3	2	1	0
F30_GPIO_Data0.*	—	—	—	—	—	—	—	GpiLedData0

Figure 54. Function \$30 per-GPIO/LED data register

This register is used to read the state of the pins programmed as GPIs. For pins programmed as a GPI, the value in this register is the same as the value on the input pin. Pins programmed as GPO have undefined values.

For pins programmed as an LED, this register has an LED’s “ReachedTarget” status. The value in this register is set to a ‘1’ when a ramp up operation completes and has reached its maximum; it is set to a ‘0’ when a ramp down operation completes and has reached its minimum. The specific behavior is seen in the following diagram. This allows the host to determine the state of a ramping operation. Note that “rampBusy” is LedActive xor ReachedTarget.

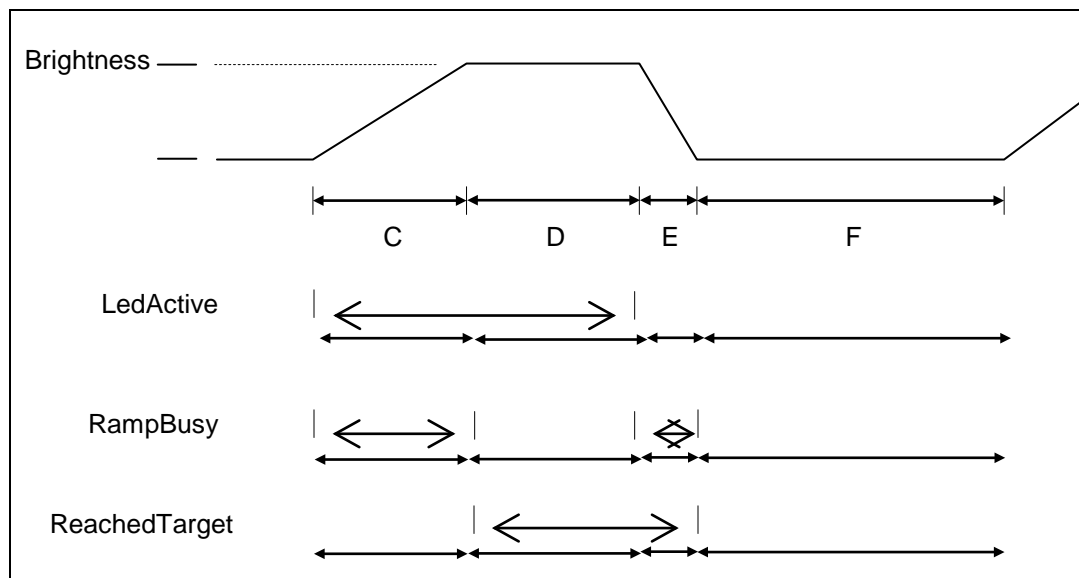


Figure 55. LED ramping status

8.4.1. Function \$30: interrupt source

Function \$30 implements one interrupt source. Function \$30 asserts an interrupt request whenever any GPI input data bit in a GPI/LED Data register changes from a ‘0’ to a ‘1’ or from a ‘1’ to a ‘0’. Because an interrupt request is a “sticky” bit, interrupt requests read as ‘1’ if any button bit has changed since the last time the data registers were read, even if the button bit has changed back to its previous value since that time. An interrupt request is asserted synchronously with the report rate.

8.5. Function \$30: command registers

Function \$30 does not implement any command registers.

8.6. Function \$30 example: complete control layout

Function \$30 contains several variable-sized control registers. Every product contains at least one variable-sized register block. The bits are mapped into the registers such that bit 0 of the GPIO/LED Select register controls the enabling of pin0, bit 1 controls pin 1, and so on. The following figure shows an example of a register layout using 11 GPIO/LED pins. For this example, *GpioLedCount* = 11, *HasMappableButtons* = 1, *HasLed* = 1, *HasGpio* = 1, *HasHaptic* = 1, *HasGpioDriverControl* = 0.

Name	7	6	5	4	3	2	1	0
F30_GPIO_Query0	—	—	HasGpio DriverControl	HasHaptic	HasGpio	HasLed	HasMappable Buttons	—
F30_GPIO_Query1	—	—	—	GpioLedCount				
F30_GPIO_Ctrl0.0	LedSel7	LedSel6	LedSel5	LedSel4	LedSel3	LedSel2	LedSel1	LedSel0
F30_GPIO_Ctrl0.1	—	—	—	—	—	LedSel10	LedSel9	LedSel8
F30_GPIO_Ctrl1	—	—	Halted	LedHalt	—	—	—	GpiDebounce
F30_GPIO_Ctrl2.0	Dir7	Dir6	Dir5	Dir4	Dir3	Dir2	Dir1	Dir0
F30_GPIO_Ctrl2.1	—	—	—	—	—	Dir10	Dir9	Dir8
F30_GPIO_Ctrl3.0	Data7	Data6	Data5	Data4	Data3	Data2	Data1	Data0
F30_GPIO_Ctrl3.1	—	—	—	—	—	Data10	Data9	Data8
F30_GPIO_Ctrl4.0	LedAct7	LedAct6	LedAct5	LedAct4	LedAct3	LedAct2	LedAct1	LedAct0
F30_GPIO_Ctrl4.1	—	—	—	—	—	LedAct10	LedAct9	LedAct8
F30_GPIO_Ctrl5.0	RampPeriodA							
F30_GPIO_Ctrl5.1	RampPeriodB							
F30_GPIO_Ctrl5.2	RampPeriodC							
F30_GPIO_Ctrl5.3	RampPeriodD							
F30_GPIO_Ctrl5.4	RampPeriodE							
F30_GPIO_Ctrl5.5	RampPeriodF							
F30_GPIO_Ctrl6.0	Pattern				Brightness			
F30_GPIO_Ctrl6.1	Pattern				Brightness			
F30_GPIO_Ctrl6.2	Pattern				Brightness			
...								
F30_GPIO_Ctrl6.8	Pattern				Brightness			
F30_GPIO_Ctrl6.9	Pattern				Brightness			
F30_GPIO_Ctrl6.10	Pattern				Brightness			
F30_GPIO_Ctrl7.0	OpenDrain	ButtonPolarity	Valid	CapacitiveButtonNumber				
F30_GPIO_Ctrl7.1	OpenDrain	ButtonPolarity	Valid	CapacitiveButtonNumber				
F30_GPIO_Ctrl7.2	OpenDrain	ButtonPolarity	Valid	CapacitiveButtonNumber				
....								
F30_GPIO_Ctrl7.8	OpenDrain	ButtonPolarity	Valid	CapacitiveButtonNumber				
F30_GPIO_Ctrl7.9	OpenDrain	ButtonPolarity	Valid	CapacitiveButtonNumber				
F30_GPIO_Ctrl7.10	OpenDrain	ButtonPolarity	Valid	CapacitiveButtonNumber				
F30_GPIO_Ctrl8.0	HapticEn7	HapticEn6	HapticEn5	HapticEn4	HapticEn3	HapticEn2	HapticEn1	HapticEn0
F30_GPIO_Ctrl8.1	—	—	—	—	—	HapticEn10	HapticEn9	HapticEn8
F30_GPIO_Ctrl9	Haptic Duration							
F30_GPIO_Data0.0	GpiLedData7	GpiLedData6	GpiLedData5	GpiLedData4	GpiLedData3	GpiLedData2	GpiLedData1	GpiLedData0
F30_GPIO_Data0.1	—	—	—	—	—	GpiLedData10	GpiLedData9	GpiLedData8

Figure 56.

8.7. Function \$30 examples: query bits

When a particular firmware is built, the query bits are fixed by the features included in the build. The subsections below describe some of the possible configurations.

8.7.1. Function \$30 example: both GPIOs and LEDs

For this example, HasLed is '1', HasGpio is '1', and HasGpioDriverControl is '1'.

Name	7	6	5	4	3	2	1	0
F30_GPIO_Ctrl0.*	LedSel7	LedSel6	LedSel5	LedSel4	LedSel3	LedSel2	LedSel1	LedSel0
F30_GPIO_Ctrl1	—	—	Halted	LedHalt	—	—	—	GpiDebounce
F30_GPIO_Ctrl2.*	Dir7	Dir6	Dir5	Dir4	Dir3	Dir2	Dir1	Dir0
F30_GPIO_Ctrl3.*	Data7	Data6	Data5	Data4	Data3	Data2	Data1	Data0
F30_GPIO_Ctrl4.*	LedAct7	LedAct6	LedAct5	LedAct4	LedAct3	LedAct2	LedAct1	LedAct0
F30_GPIO_Ctrl5.*	RampPeriodA							
F30_GPIO_Ctrl6.*	Pattern-Brightness-for-LEDs =or= STRPU-STRPD-SPCTRL-for-GPIOs							

Figure 57

8.7.2. Function \$30 example: LEDs only

For this example, HasLed is '1' and HasGpio is '0'.

Name	7	6	5	4	3	2	1	0
F30_GPIO_Ctrl1	—	—	Halted	LedHalt	—	—	—	GpiDebounce
F30_GPIO_Ctrl4.*	LedAct7	LedAct6	LedAct5	LedAct4	LedAct3	LedAct2	LedAct1	LedAct0
F30_GPIO_Ctrl5.*	RampPeriod*							
F30_GPIO_Ctrl6.*	Pattern				Brightness			

Figure 58

8.7.3. Function \$30 example: GPIOs only, with driver control

For this example, HasLed is '0', HasGpio is '1', and HasGpioDriverControl is '1'.

Name	7	6	5	4	3	2	1	0
F30_GPIO_Ctrl1	—	—	Halted	LedHalt	—	—	—	GpiDebounce
F30_GPIO_Ctrl2.*	Dir7	Dir6	Dir5	Dir4	Dir3	Dir2	Dir1	Dir0
F30_GPIO_Ctrl3.*	Data7	Data6	Data5	Data4	Data3	Data2	Data1	Data0
F30_GPIO_Ctrl6.*	—	STRPU	—	STRPD	SPCTRL			—

Figure 59

8.7.4. Function \$30 example: GPIOs only, without driver control

For this example, HasLed is '0', HasGpio is '1', and HasGpioDriverControl is '0'.

Name	7	6	5	4	3	2	1	0
F30_GPIO_Ctrl1	—	—	Halted	LedHalt	—	—	—	GpiDebounce
F30_GPIO_Ctrl2.*	Dir7	Dir6	Dir5	Dir4	Dir3	Dir2	Dir1	Dir0
F30_GPIO_Ctrl3.*	Data7	Data6	Data5	Data4	Data3	Data2	Data1	Data0

Figure 60

9. Function \$32: Timer

Function \$32 implements a general purpose timer suitable for timing a variety of events in an RMI device. The timer counting period is nominally one count every 10 milliseconds. The timer accuracy is device-dependent, but it is anticipated that a typical timer implemented in an RMI device will be accurate within 15 percent. If desired, the timer can be configured to generate an interrupt request to the host under a variety of circumstances.

9.1. Function \$32: query registers

9.1.1. F32_Timer_Query0: Timer properties query

Timer Function \$32 implements a single query register, the General Properties query.

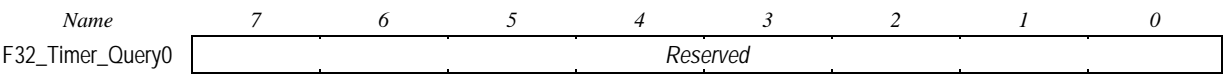


Figure 61. Function \$32 query register

The contents of this register are reserved for future use.

9.2. Function \$32: control registers

The two control registers associated with the timer are defined to be part of a write-coherency region (see section 2.6 for more information on coherency regions). This means that special access considerations apply when writing the Timer control registers. There are exactly two permissible ways to write these registers:

1. Write Timer_Ctrl0, then write Timer_Ctrl1.
In this case, the write to Timer_Ctrl0 is deferred until the data for Timer_Ctrl1 is written by the host. Both control registers are updated simultaneously, as soon as Timer_Ctrl1 is written.
2. Write Timer_Ctrl1 only.
In this case, Timer_Ctrl1 is updated, while Timer_Ctrl0 remains unchanged. This access method allows a host to update just the *TimerMode* or *TimerEnable* field without having to write the *MatchCount* 7:0. Writing Timer_Ctrl1 will always update *MatchCount* 9:8, even if Timer_Ctrl0 was not written in the previous transfer.

9.2.1. F32_Timer_Ctrl0, 1: timer match count, mode, and enable

The Timer control registers are defined as follows:

Name	7	6	5	4	3	2	1	0
F32_Timer_Ctrl0	MatchCount 7:0							
F32_Timer_Ctrl1	TimerMode		—	TimerEnable		—	MatchCount 9:8	

Figure 62. Function \$32 control registers

MatchCount 7:0 (F32_Timer_Ctrl0, bits 7:0)

This field holds the low-order 8 bits of the 10-bit *MatchCount*. The reset default *MatchCount* is '0'.

MatchCount 9:8 (F32_Timer_Ctrl1, bits 1:0)

This field holds the most significant two bits of the 10-bit *MatchCount*. The reset default *MatchCount* is '0'.

TimerEnable (F32_Timer_Ctrl1, bits 4:3)

This field controls how the timer is enabled and disabled. The reset default *TimerEnable* mode is '00', Disable Counting. Writing to this register clears *TimerCount* 9:0 to 0 and unlocks the timer.

TimerEnable = '00': *Disable Counting*.

In this mode, counting is disabled.

TimerEnable = '01': *Enable Counting*.

In this mode, counting is enabled. The *TimerCount* increments at its natural counting rate unless the count has been locked due to a match event in *TimerMode* '11', Stop On Match.

TimerEnable = '10': *Finger Timer*.

This control field only applies to RMI devices that support finger sensing. For those devices, timer counting is stopped while no fingers are present. When a finger arrives on any finger sensor, the timer count is cleared to 0, the timer is unlocked, and counting begins. Counting continues until no fingers are present, or until the timer count is locked by a match in *TimerMode* '11', Stop On Match.

TimerEnable = '11': *Reserved.*

This mode is reserved for future use.

TimerMode (*F32_Timer_Ctrl1*, bits 7:6)

This field defines how the timer operates. The reset default *TimerMode* is '00', Wrap mode.

TimerMode = '00': *Wrap Mode.*

In this mode, the timer count wraps to 0 after reaching the maximum count of 1023.

TimerMode = '01': *Reset On Read.*

As in Wrap mode, the timer count wraps to 0 after reaching the maximum count of 1023. In addition, *TimerCount* 9:0 is cleared to 0 whenever the Timer data 0 register is read.

TimerMode = '10': *Reset On Match.*

The *TimerCount* 9:0 is cleared to 0 whenever the *TimerCount* 9:0 matches the *MatchCount* 9:0.

TimerMode = '11': *Stop On Match.*

When the *TimerCount* 9:0 matches the *MatchCount* 9:0, the *Match* bit in Timer data 0 is set, and the timer is locked (counting stops).

The rest of the bits in this register are reserved.

Special access considerations apply to writing these registers. In particular, *F32_Timer_Ctrl0* can't be written by itself.

9.3. Function \$32: data registers

The Timer data registers contain the current *TimerCount*, along with some status flags. All data registers are read-only.

The two data registers associated with the Timer belong to a read-coherency region (see section 2.6 for more information on coherency regions). Special access considerations apply when reading the Timer data registers. To read *TimerCount* 9:0 in a coherent fashion, a host must read F32_Timer_Data0, then F32_Timer_Data1. F32_Timer_Data0 can't be read by itself, without reading F32_Timer_Data1 as the next register access to the device.

F32_Timer_Data1 can be read by itself without reading F32_Timer_Data0. This allows a host to do things like test the *Match* bit without having to read the entire timer count.

9.3.1. F32_Timer_Data0, 1: Timer count, match, and run

The bits of these registers are defined as follows:

Name	7	6	5	4	3	2	1	0
F32_Timer_Data0	TimerCount 7:0							
F32_Timer_Data1	Match	Run	—	—	—	—	TimerCount 9:8	

Figure 63. Function \$32 data registers

TimerCount 7:0 (F32_Timer_Data0, bits 7:0)

This field holds the least significant eight bits of the 10-bit *TimerCount*. The reset default *TimerCount* is '0'.

TimerCount 9:8 (F32_Timer_Data1, bits 1:0)

This field holds the most significant two bits of the 10-bit *TimerCount*. The reset default *TimerCount* is '0'.

Run (F32_Timer_Data1, bit 6)

If '0', the Timer is stopped. If '1', the Timer is counting. The reset default value is '0'.

Match (F32_Timer_Data1, bit 7)

When *TimerMode* is '10' (Reset On Match) or '11' (Stop On Match), the *Match* bit indicates the following:

- If '0', the *TimerCount* has not reached the *MatchCount* since this register was last read.
- If '1', the *TimerCount* reached the *MatchCount* since this register was last read. A Timer interrupt request is generated whenever the *Match* bit changes from '0' to '1'.

The *Match* bit is always '0' when *TimerMode* is '00' (Wrap Mode) or '01' (Reset On Read).

The *Match* bit is cleared whenever the Timer data registers are read. See the Timer data register description for special access considerations when reading the Timer data registers.

The reset default value is '0'.

9.3.2. Function \$32: interrupt source

The data registers defined by Function \$32 are associated with a single interrupt request source, called the Timer interrupt request. Any product that includes Function \$32 allocates a timer interrupt request bit in the Interrupt Status register (see section 3.3.2), and a timer interrupt enable bit in the Interrupt Enable register (see section 3.2.2). The position of the timer interrupt bits in those registers is product-specific; consult the product-specific documentation to determine their location.

The timer interrupt request bit in the Interrupt Status register is asserted on every timer tick where the *Match* bit has the value '1'. The *Match* bit can be cleared by reading Timer data register 1.

9.4. Function \$32: command registers

Function \$32 does not implement any command registers.

10. Function \$34: Flash memory management

Function \$34 implements all functionality related to flash memory. Flash programming can be used to upgrade the user interface (UI) firmware in the field or to alter the configuration of the RMI4 device.

In addition to flash erase and programming operations, Function \$34 provides fundamental integrity assurance for that flash memory and the firmware and configuration it contains. Function \$34 provides the following features:

- Boot-time integrity check for the UI firmware and its configuration.
- Mechanism to bypass execution of UI firmware even if it passes the integrity check.
- Firmware and Configuration erase and reprogram function.
- Mechanism to recover from interrupted or failed erase/reprogram attempts.

10.1. Overview

10.1.1. Non-Volatile Memory Organization

Figure 64 shows the basic storage methodology for the device firmware:

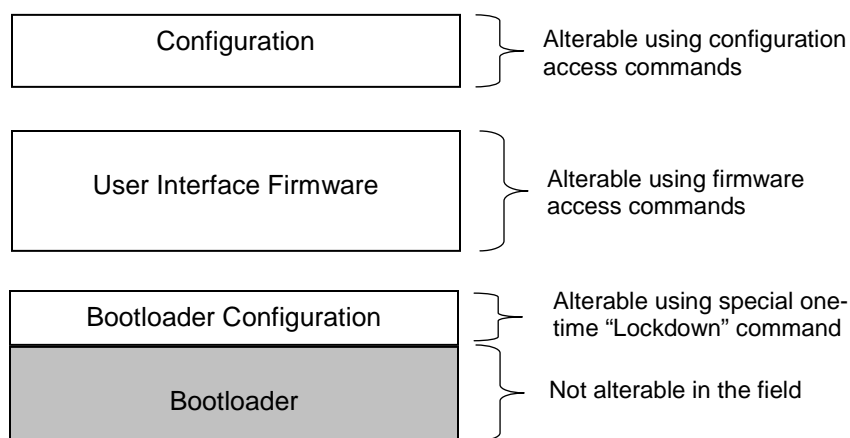


Figure 64. Firmware storage methodology

Configuration

The configuration space stores the default values of the device's control registers. The bootloader firmware provides a mechanism to erase and reprogram this space.

Because an existing configuration may not be valid for a new firmware revision, any update to the UI firmware should be followed by an update of the configuration space.

User interface firmware

The UI firmware space contains the firmware that implements the primary function of the device. The bootloader firmware provides a mechanism to erase and reprogram this space.

UI firmware images are provided by Synaptics in an encrypted form to ensure they can only be executed on an appropriate device.

It is not possible to erase the UI firmware space without also erasing the configuration space.

Bootloader firmware / Bootloader configuration

This space contains the bootloader firmware. The bootloader firmware:

- checks the integrity of the UI firmware space, and
- provides the ability to re-flash a new UI or configuration area.

To guarantee that a device can never be unrecoverably corrupted during a firmware update operation, this space is not field-programmable via Function \$34. However, in order to implement the bootloader Lockdown command, a bootloader will permit a small, one-time change confined to the configuration area of the bootloader space. Because the bootloader space is not field-erasable, the lockdown operation is permanent.

The size of these spaces is product specific and can be determined by querying the Function \$34 registers.

10.1.2. Modality

Function \$34 flash programming functions are mutually exclusive with most other RMI4 functions. Two modes determine which functions are available:

- **UI mode:** This is the default RMI4 mode, but in Function \$34 only one command is operable in this mode: Enable Flash Programming, which is used to enter Flash Programming mode. A device is in UI mode when the *Flash Programming En* bit is '0'. A Function \$01 Reset command is used to end Flash Programming mode.

Note: In addition to the host initiated entry to Flash Programming mode it is possible for the ASIC itself to initiate entry to the Flash Programming mode as part of the recovery mechanism for programming failures due to interruption or other causes. See section 3.3.1 for more information on Flash CRC errors.

- **Bootloader mode:** All other Function \$34 commands only function in this mode. A device is in Bootloader mode (also sometimes known as Flash Programming mode) when the *Flash Programming En* bit is '1'.

When Flash Programming is enabled, the device is in Bootloader mode and the normal operation of the device is disabled. Only the following subset of functions is available:

- *Page Description Tables*
The Page Description tables accurately reflect the reduced functionality offered in Bootloader mode. Only Functions \$01 and \$34 are visible.



Important: The data in the Page Description tables may be different from the Page Description tables operating in UI mode. Hosts should re-scan the Page Description tables when both entering and exiting Bootloader mode.

- *Page Select Register*
The Page Select Register always selects Page \$00.



Important: Writes to the Page Select Register are ignored.

- *Function \$01: Control Register 1 – Interrupt Enable* register.
Only the Function \$34 interrupt enable bit is writable, all others are forced to 0, disabling those sources.
- *Function \$01: Data Register 1 – Device Status* register.
The *Flash Prog* bit is set and the *Status Code* field indicates the reason that the bit is set.
- *Function \$01: Data Register 1 – Interrupt Status* register
Only the Function \$34 interrupt source asserts.
- *Function \$34:*
All of the Function \$34 registers function as documented in this specification.

10.2. Function \$34: query registers

10.2.1. F34_Flash_Query0, 1: Bootloader ID query

This register reports the unique 16-bit Bootloader ID. This ID is used to identify the bootloader revision. For example, a Version 3 bootloader would set *Bootloader ID 0* to ‘V’ and *Bootloader ID 1* to ‘3’.

Name	7	6	5	4	3	2	1	0
F34_Flash_Query0	Bootloader ID 0							
F34_Flash_Query1	Bootloader ID 1							

Figure 65. Function \$34 Bootloader ID query registers

10.2.2. F34_Flash_Query2: Flash Properties query

This byte contains bits that describe whether the RMI product has various optional properties.

Name	7	6	5	4	3	2	1	0
F34_Flash_Query2	Reserved						Unlocked	RegMap

Figure 66. Function \$34 Flash Properties query register

Each property bit is ‘1’ if the product has the associated property, or ‘0’ if the product does not have the associated property. Reserved property bits report as ‘0’, but they may report as ‘1’ in devices that comply with a future version of flash functions.

RegMap Query (F34_Flash_Query2, bit 0)

This bit always reports as 1.

Unlocked (F34_Flash_Query2, bit 1)

This command has no meaning in UI mode. In Bootloader mode, a device can be in either a locked or an unlocked state:

- A ‘1’ indicates that the device is currently in the generic, unlocked state. A device in this state is capable of being locked down.
- A ‘0’ in this bit location indicates that either the device has already been locked down, or that it is a legacy device. In either case, the generic aspects of device operation are fixed and cannot be locked down.

10.2.3. F34_Flash_Query3, 4: Block Size query

The Block Size indicates the number of bytes in one data block. When programming the firmware, the data should be broken into blocks of this size and each block programmed individually.

Name	7	6	5	4	3	2	1	0
F34_Flash_Query3	Block Size (bits7:0)							
F34_Flash_Query4	Block Size (bits15:8)							

Figure 67. Function \$34 Block Size query registers

10.2.4. F34_Flash_Query5, 6: Firmware Block Count query

The *Firmware Block Count* indicates the number of blocks in a firmware image. $Block\ Size * Firmware\ Block\ Count = \text{total number of bytes in a firmware image area.}$

Name	7	6	5	4	3	2	1	0
F34_Flash_Query5								Firmware Block Count (bits7:0)
F34_Flash_Query6								Firmware Block Count (bits15:8)

Figure 68. Function \$34 Firmware Block Count query registers

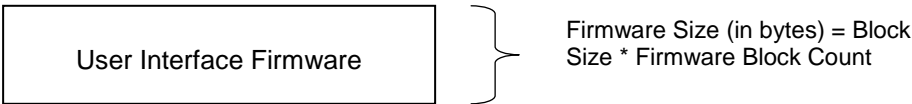


Figure 69. Firmware Size

10.2.5. F34_Flash_Query7, 8: Configuration Block Count query

The *Configuration Block Count* indicates the number of blocks in a configuration image. $Block\ Size * Configuration\ Block\ Count = \text{total number of bytes in a configuration image.}$

Name	7	6	5	4	3	2	1	0
F34_Flash_Query7								Configuration Block Count (bits: 7:0)
F34_Flash_Query8								Configuration Block Count (bits 15:8)

Figure 70. Function \$34 Configuration Size query registers

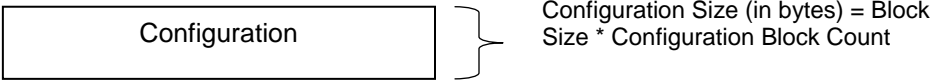


Figure 71. Configuration Size

10.3. Function \$34: data registers

Function \$34's data registers specify block numbers, report block data, and control all aspects of flash update operations.

Name	7	6	5	4	3	2	1	0
F34_Flash_Data0	BlockNum 7:0							
F34_Flash_Data1	BlockNum 15:8							
F34_Flash_Data2.*	Block Data							
F34_Flash_Data3	Program Enabled	Flash Status			Flash Command			

Figure 72. Function \$34 data registers

10.3.1. F34_Flash_Data0,1: Block Number registers

The Block Number registers are used to specify which Flash Block is accessed by flash commands. These registers should be written with a block number prior to issuing any of the following commands:

- Read Firmware Block
- Write Firmware Block
- Read Configuration Block
- Write Configuration Block

The Block Number registers automatically increment after each operation.

10.3.2. F34_Flash_Data2.*: Block Data registers

The Block Data registers are used to read or write the data for a block. F34_Flash_Data2 is a replicated register (see section 2.3.6 for a definition of these registers). Each implementation has at least one of these registers, but the exact count is implementation-specific. The value obtained from the F34_Flash_Query3,4 'block size' registers represents the exact number of F34_Flash_Data2 registers (number of registers = n , where n is Block Size).

The Block Data registers must be written with data prior to a block write operation. At the completion of a block read operation, the Block Data registers contain the results of the read operation.

Prior to executing an *Erase All*, *Erase Configuration* or *Enable Flash Programming* operation, the first two registers must be written with the Bootloader ID, which acts as a key to enable the operation.

10.3.3. F34_Flash_Data3: Flash control/status register

This register is used to control all aspects of flash update operation.

Name	7	6	5	4	3	2	1	0
F34_Flash_Data3	Program Enabled	Flash Status			Flash Command			

Figure 73. Function \$34 control/status register

Flash Command (F34_Flash_Data3, bits 3:0)

When this field is written the requested operation is performed. This field is not writable while the previous operation is still in progress – this field is only writable when the current value is \$0.

When a requested operation is completed this field is set back to \$0 by the device and a Function \$34 interrupt is asserted. All commands that can be used to read, erase, or program the flash can only be issued when the device is in a ‘Flash Programming Enabled’ state.

Table 5. Function \$34 Flash command values

Value	Meaning	Available in UI Mode?	Available in Bootloader mode?	Requires key before issuing command?
\$0	Idle – No command is active	Yes	Not applicable	Not applicable
\$1	<i>Reserved</i>			
\$2	Write Firmware Block	No	Yes	No
\$3	Erase All	No	Yes	Yes
\$4	Write Lockdown Block	No	Yes	No
\$5	Read Configuration Block	No	Yes	No
\$6	Write Configuration Block	No	Yes	No
\$7	Erase Configuration	No	Yes	Yes
\$8-\$E	<i>Reserved</i>			
\$F	Enable Flash Programming	Yes	No	Yes

Flash Command \$00: Idle

There are no Function \$34 commands in process and the device is ready to receive a Function \$34 command.

*Flash Command \$01: Reserved.**Flash Command \$02: Write Firmware Block.*

This command is issued after a Block Number is written to the Block Number registers and the data block is written to the Block Data registers. This command actually writes the block into the UI Firmware space. The UI Firmware space must have first been erased with command \$03 prior to writing the firmware blocks. An attempt to write to a non-erased firmware block will result in an error 5 – “*Block Not Erased*”.

Flash Command \$03: Erase All.

This command clears both the UI Firmware space and the Configuration space. Use it before programming (writing) the new image with command \$02. Because an existing configuration is likely invalid for a new firmware revision, this command also clears the configuration area, which should be programmed with the new configuration associated with the new firmware revision.



Important: Prior to executing this command, the host must write the Bootloader ID to the Block Data registers as a “key value” to reduce the possibility of an accidental erasure.

Flash Command \$04: Write Lockdown Block.

This command writes the lockdown blocks. It is anticipated that the number of blocks to be written to lock a device down will be very small (about three blocks in total). The number of blocks is defined by the image file itself. For information about the RMI4 image file, see the *RMI4 Bootloader Procedures* application note (PN: 506-000221-01).

This command is only processed if the *Unlocked* bit is in the '1' (unlocked) state. The lockdown information can't be erased once it is written. The lockdown operation is a one-time affair. The changes caused by the lockdown operation will not take effect until the device is either reset or an 'Enable Flash Programming' command is issued.

The actual lockdown operation is performed after the final lockdown block is written and before the status code is returned. Lockdown blocks written before the final lockdown block may report "success" as they buffer the lockdown information.

When the final block is written, the device will report the overall success or failure based on the following table:

Table 6

<i>State</i>	<i>Result code</i>
Unlocked , and lockdown data is valid	0 ("Success")
Unlocked , but lockdown data is invalid	2 ("Flash Programming Not Enabled / Bad Command")
Unlocked , but block number is invalid	3 ("Invalid Block Number")
Locked , and block number is valid	4 ("Block Not Erased")
Locked , but block number is not valid	3 ("Invalid Block Number")

The response depends on the bootloader's current lockdown state when it receives the command, and the state of the block number and the lockdown data itself.

Flash Command \$05: Read Configuration Block.

This command reads a block of data from the configuration space and places it into the Block Data registers to be read by the host. This command is issued after a block number is written to the Block Number register. After the command has completed, the block read data is available to be read from the Block Data registers.

Flash Command \$06: Write Configuration Block.

This command is issued after a block number is written to the Block Number register and the data block is written to the Block Data registers. This command actually writes the block into the Configuration space.

The Configuration space must have first been erased with command \$07 prior to writing the firmware blocks. An attempt to write to a non-erased configuration block will result in an error 5 – "Block Not Erased".

Flash Command \$07: Erase Configuration.

This command clears the Configuration space. It should be used prior to programming (writing) the new configuration, one block at a time, with command \$06.



Important: Prior to executing this command, the host must write the Bootloader ID to the Block Data registers as a “key value” to reduce the possibility of an accidental erasure.

Flash Command \$0F: Enable Flash Programming.

The Enable Flash Programming command is used to initiate flash programming operations. Prior to the execution of this command the only Function \$34 registers that are functional are the query registers and the command register in which only this command is supported.

As a side-effect of this command, the Function \$01 Interrupt Enable (*F01_RMI_Ctrl1*) register is forced to disable all interrupts except for the Function \$34 interrupt, which is forced enabled.

When flash programming is complete, the host must issue an RMI Reset command to put the device back into normal operating mode.



Important: Prior to executing this command, the host must write the Bootloader ID to the Block Data registers as a “key value” to reduce the possibility of accidentally entering Flash Programming mode.

Attempts to execute this command when Flash Programming mode is already enabled are treated as a no-operation and always return Success.

Flash Status (F34_Flash_Data3, bits 6:4)

This field, at the completion of a flash operation, indicates the success or failure of the operation. Writes to this field are ignored.

Table 7. Function \$34 Flash Status values

Value	Meaning
0	Success
1	Reserved
2	Flash Programming Not Enabled/Bad Command
3	Invalid Block Number
4	Block Not Erased
5	Erase Key Incorrect
6	Unknown Erase/Program Failure
7	Device has been reset

Program Enabled (F34_Flash_Data3, bit 7)

This read-only bit indicates that flash programming has been enabled by an Enable Flash Programming command, or automatically due to a Flash CRC error during boot. If a CRC error is detected, the Function \$01 Flash Prog field (*F01_RMI_Data0*, bit 6) is set and the specific cause of the error can be found in the Function \$01 Status Code (*F01_RMI_Data0*, bits 3:0). This bit is set to ‘1’ for Bootloader mode, and ‘0’ (the default) for UI mode.

When this bit is set, only a very small subset of the RMI interface is available: the Function \$34 registers and the Function \$01 registers required to manage the Function \$34 interrupt. See section 10.1.2 for more information. This bit is cleared only when a Function \$01 Reset command is executed.

10.3.4. Function \$34: interrupt source

Function \$34 is associated with a single interrupt request source, called the Flash interrupt request. Any product that includes Function \$34 allocates a Flash interrupt request bit in the Interrupt Status register (see section 3.3.2), and a Flash interrupt enable bit in the Interrupt Enable register (see section 3.2.2). The position of the Flash interrupt bits in those registers is product-specific; consult the product-specific documentation to determine their location.

10.4. Flash programming procedures

These are the suggested procedures for common flash functions. To fully reflash the UI image and configuration, the operations in sections 10.4.2 through 10.4.5 must complete successfully.

10.4.1. Bootloader mode and ATTN

An RMI device may be unable to respond to physical layer communication while executing bootloader commands. Because of this, an RMI device will assert the ATTN signal to indicate that a command has completed, and that the physical layer interface is active again.

Once the host observes ATTN after a command completes, the interrupt should be cleared in the normal fashion by reading the Interrupt Status register before proceeding with the next bootloader operation. Some RMI devices may not implement an ATTN signal. In addition, there may be hosts that do not have the ability to read the state of ATTN, even if the RMI device supports it. In those cases, there are two options to determine when the command has completed.

The first option is to poll the device, knowing that the physical layer may be inactive while the device is busy. In the case of devices that implement an I²C or SMBus physical layer, the device will NAK any transfers sent to it during the period where the command is being executed. As soon as the command completes, the device will respond normally. In the case of devices implementing an SPI interface, the device will respond with the default state of the data bus until such time as the command completes. In either case, the host driver needs to recognize those situations as representing a ‘command in progress’, as opposed to being a physical layer error.

The second option is to insert host-generated time delays after issuing a bootloader command. While simple, this approach is not time efficient, so the reflashing operation takes much longer than it would if the ATTN handshake was used. The time delay after issuing a flash command is module-specific.

10.4.2. Enable flash programming

Before any actual erase or program operations can be performed, the Flash Programming mode must be enabled. Follow this procedure to enable flash programming:

1. Read the Bootloader ID registers (F34_Flash_Query0 and 1) to obtain the key.
2. Write the Bootloader ID key data back to the first two Block Data registers (F34_Flash_Data2.0 and F34_Flash_Data2.1).
3. Issue a Flash Program Enable (\$0F) command to the *Flash Command* (F34_Flash_Data3, bits 3:0) field.

4. Wait for ATTN. The *Flash* interrupt request bit will be '1'.
5. Read the *Flash Command* field in the F34_Flash_Data3 register. The result should be \$80: Flash Command = Idle (\$0), Flash Status = Success (\$0), Program Enabled = Enabled (1). Any other value indicates an error.
6. Rescan the Page Description table.



Important: The nature of the changes in the new image may result in the device registers “moving around” in the RMI register map. Rescanning the Page Description table ensures that it reflects any relocations that have occurred due to the change from Bootloader mode to Flash Programming mode.

10.4.3. Program the firmware image

After the device is in Flash Programming mode, program the firmware image by following this procedure:

1. Read the Bootloader ID registers (F34_Flash_Query0 and F34_Flash_Query1) to obtain the key.
2. Write the Bootloader ID key data back to the first two Block Data registers (F34_Flash_Data2.0 and F34_Flash_Data2.1).
3. Issue the Firmware and Configuration Erase (\$03) command to the *Flash Command* (F34_Flash_Data3, bits 3:0) field.
4. Wait for ATTN. The *Flash* interrupt request bit will be '1'.
5. Read the *Flash Command* field in the F34_Flash_Data3 register. The result should be \$80: Flash Command = Idle (\$0), Flash Status = Success (\$0), Program Enabled = Enabled (1). Any other value indicates an error.
6. For each block in the firmware image, starting at Block 0 and proceeding consecutively:
 - a) Write the Block Number (F34_Flash_Data0 and 1) register.

Note: Because the Block Number register auto-increments after each Write Firmware Block operation, this step is only required before writing Firmware Block 0.
 - b) Write the Block Data starting at F34_Flash_Data2. Fill the next *n* registers with the data to be programmed.
 - c) Issue the Write Firmware Block (\$02) command to the *Flash Command* (F34_Flash_Data3, bits 3:0) field.

Note: Steps a, b, and c can be performed in a single write transaction because of the way registers in an RMI register map are organized.
 - d) Wait for ATTN. The *Flash* interrupt request bit will be '1'.
7. Program the configuration image.

10.4.4. Program the configuration image

After the device is in Flash Programming mode and the configuration area has been erased by either an Erase Firmware and Configuration Areas command or an Erase Configuration Area command, program the configuration image by following this procedure:

1. For each block in the configuration image, starting at Block 0 and proceeding consecutively:
 - a) Write the Block Number (F34_Flash_Data0 and 1) register.
 - Note:** Because the Block Number register auto-increments after each Write Configuration Block operation, this step is only required before writing Configuration Block 0.
 - b) Write the Block Data starting at F34_Flash_Data2. Fill the next *n* registers with the data to be programmed.
 - c) Issue a Write Configuration Block (\$06) command to the *Flash Command* (F34_Flash_Data3, bits 3:0) field.
 - Note:** Steps a, b, and c can be performed in a single write transaction because of the way registers in an RMI register map are organized.
 - d) Wait for ATTN. The *Flash* interrupt request bit will be '1'.
 - e) Read the *Flash Command* field in the F34_Flash_Data3 register. The result should be \$80: Flash Command = Idle (\$0), Flash Status = Success (\$0), Program Enabled = Enabled (1). Any other value indicates an error.
2. Disable Flash Programming mode to put the new configuration into effect.

10.4.5. Disable Flash Programming mode

After all flash programming operations are complete, resume normal operation by resetting the device:

1. Issue a Reset (\$01) command to the *RMI Command* (F01_RMI_Cmd0, bit 0) field. This tests the firmware image and executes it if it is valid.
2. Wait for ATTN to assert.
3. At this point, the Bootloader will perform a CRC on the UI Firmware and the Configuration space, and the *Program Enabled* bit (bit 6 of F01_RMI_Data0) will be set to '1' to reflect this.
4. Eventually, if there is no CRC failure, ATTN is de-asserted and then asserted again as the firmware enters UI mode.
5. The *Program Enabled* bit is cleared to '0' to indicate that the new firmware is valid and is executing in UI mode.
6. A host driver should re-scan the Product Description table after restarting the new UI image.



Important: Changes in the new image may cause the device registers to move to new locations in the RMI register map. Rescanning the Product Description table ensures that the host driver properly tracks any relocations.



Important: The device has not been successfully reprogrammed until step 6 is reached. A CRC failure in the UI Firmware or Configuration space will cause the device to remain in Bootloader mode with the *Program Enabled* bit set to '1' and an error code in the *Status Code* field (bits 3:0 of F01_RMI_Data0).

For more information about the suggested rules and procedures for bootloader use, see the *RMI4 Bootloader Procedures* (PN: 506-000221-01) application note.

10.5. Configuration space layout

The layout of the Configuration space is product-specific, but an RMI4 device will always contain enough information for a host to determine the exact layout of the Configuration space for a given product. The procedure for a host driver to work out the Configuration space is as follows:

- The Configuration space contains the default value for every control register belonging to every RMI function that the device supports.
- The control registers are grouped by function in the Configuration space, exactly as they are grouped by function in the RMI register map.



Important: Query register bits within an RMI function may define rules that either cause certain control registers to exist or not, or that define the size of a replicated register area. The same rules define the layout of the control registers in the Configuration space for that product. Therefore, the layout and count of control registers in the Configuration space for a given function in a given product are always identical to the layout of the control registers for that function in the RMI register map.

- The ordering of the control registers defined by each of the RMI functions in the Configuration space is defined by the ordering of the RMI functions in the PDT table.
- The placement of the control register defaults begins at offset 0 in the Configuration space.

For example, a device contains three RMI4 functions: F\$34 (Reflash), F\$01 (General Control), and F\$19 (0-D Capacitive Buttons). The Page Description table shows that the RMI functions appear in the order F\$34, F\$01, and F\$19. This means that the default values for the control registers will be placed into the Configuration space in that same order: F\$34, F\$01, and then F\$19:

1. F\$34 does not define any RMI control registers. As a result, F\$34 allocates no Configuration space.
2. F\$01 is the next function to be processed. F\$01 defines some RMI control registers: F01_RMI_Ctrl0 and F01_RMI_Ctrl1.*. Because F01_RMI_Ctrl1.* is a replicated register (see section 2.3.6), the actual number of those registers must be determined as described in section 3.3.2. For the purposes of this example, the device defines a single F01_RMI_Ctrl1 register. This means that F\$01 will allocate the first two bytes of the Configuration space to define the defaults for these two registers.
3. The F\$19 control registers are placed into the Configuration space next. There are four sets of replicated control registers associated with F\$19. For the purposes of this example, the device defines an F\$19_Btn_Query1 *ButtonCount* of 4, just as the example shown in section 7.2. The calculations defined in section 7.2.1 indicate that for a *ButtonCount* of 4, F\$19 will define a total of 11 control registers.

The resulting assignments in the configuration space for this device are shown in the following table.

Table 8

<i>Offset within Config area</i>	<i>RMI Function</i>	<i>Register ID</i>	<i>Register Name</i>
\$00	F\$01	F01_RMI_Ctrl0	RMI Device Control
\$01	F\$01	F01_RMI_Ctrl1.0	Interrupt Enable
\$02	F\$19	F19_Btn_Ctrl0	F\$19 General Control
\$03	F\$19	F19_Btn_Ctrl1.0	F\$19 Button Interrupt Enables
\$04	F\$19	F19_Btn_Ctrl2.0	F\$19 Single Button Participation
\$05	F\$19	F19_Btn_Ctrl3.0	F\$19 Sensor Map 0
\$06	F\$19	F19_Btn_Ctrl3.1	F\$19 Sensor Map 1
\$07	F\$19	F19_Btn_Ctrl3.2	F\$19 Sensor Map 2
\$08	F\$19	F19_Btn_Ctrl3.3	F\$19 Sensor Map 3
\$09	F\$19	F19_Btn_Ctrl4.0	F\$19 Reserved 0
\$0A	F\$19	F19_Btn_Ctrl4.1	F\$19 Reserved 1
\$0B	F\$19	F19_Btn_Ctrl4.2	F\$19 Reserved 2
\$0C	F\$19	F19_Btn_Ctrl4.3	F\$19 Reserved 3
\$0D - \$FB	Warning: All space after the last function is reserved for Synaptics use. <i>Do not alter any data values in this space!</i>		
\$FC	Configuration Checksum Byte 0 (least significant byte)		
\$FD	Configuration Checksum Byte 1		
\$FE	Configuration Checksum Byte 2		
\$FF	Configuration Checksum Byte 3 (most significant byte)		

The checksum is a 32-bit Fletcher checksum, stored in little-endian fashion. (See “Fletcher’s Checksum” on Wikipedia for more information on calculating a Fletcher-32 checksum.) The checksum is always calculated over bytes \$00 through \$FB of the Configuration space, regardless of the product.

If any changes are made to the Configuration space, a new checksum must be calculated before writing the new contents back to the Configuration space. A host should always rewrite the entire Configuration space (including the new checksum) when performing an update.

11. Function \$36: Auxiliary ADC

Function \$36 implements controls for an analog-to-digital voltage conversion (ADC) function. This can be used for converting external voltages from other types of sensor and reporting these over RMI along with data from the capacitive sensor.

11.1. Function \$36: query register

The ADC Function \$36 implements a single query register, the General Properties query.

Name	7	6	5	4	3	2	1	0
F36_ADC_Query0	—	—	—	—	—	—	NumChannels	

Figure 74. Function \$36 query register

The bits in register F36_ADC_Query0 are defined as follows:

NumChannels (F36_ADC_Query0, bits 1:0)

‘00’: One conversion channel available.

‘01’: Two conversion channels available.

11.2. Function \$36: control register

The control register determines the mode of operation of the ADC channels.

Name	7	6	5	4	3	2	1	0
F36_ADC_Ctrl0	—	—	—	—	—	—	Enable1	Enable0

Figure 75. Function \$36 control register

The fields in this control register are defined as follows:

Enable0 (F36_ADC_Ctrl0, bit 0)

Enables continuous conversion by ADC channel 0.

Enable1 (F36_ADC_Ctrl0, bit 1)

Enables continuous conversion by ADC channel 1.

When enabled, continuous conversion takes place at the report rate only while the chip is fully awake. Specifically, the chip is fully awake when the capacitance sensors detect a finger, any RMI register is written to, or the No-Sleep bit is set. When the chip goes into Doze mode, ADC conversions are not performed regardless of the F36_ADC_Ctrl0 register setting. The F36 command register provides a means to force an ADC conversion during Doze mode.

11.3. Function \$36: data registers

The data registers contain the results of the conversions. There will be a minimum of one data register but the number available on a given sensor module corresponds to the number of channels available as reported by the query register. The example below is for a sensor module with two channels.

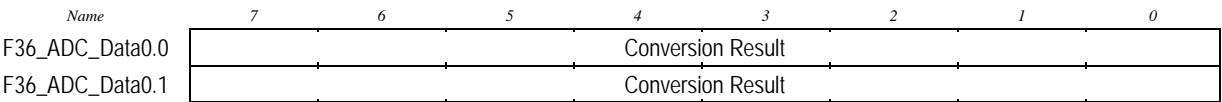


Figure 76. Function \$36 data registers, two-channel example

ConversionResult (F36_ADC_Data0., bits 7:0)*

Reports the result of the conversion.

11.4. Function \$36: command registers

The command register initiates conversions by the ADC channels.

Name	7	6	5	4	3	2	1	0
F36_ADC_Cmd0	—	—	—	—	—	—	Conv1	Conv0

Figure 77. Function \$36 command register

The fields in this register are defined as follows:

Conv0 (F36_ADC_Cmd0, bit 0)

Start conversion on ADC channel 0.

Conv1 (F36_ADC_Cmd0, bit 1)

Start conversion on ADC channel 1.

Note that writing to these bits provides a means to force an ADC conversion during Doze mode. When the ADC conversion completes, the respective command bit is automatically cleared to '0'. The clearing of the bit can thus be used as an indicator to gauge when conversions are complete. Alternately, an interrupt pin via F\$01 is also available to notify the host when an ADC conversion completes. Keep in mind that whenever continuous ADC conversion is enabled, an interrupt will occur at the report rate while the chip is fully awake using the alternate method.

11.5. Function \$36: interrupt source

The data registers defined by Function \$36 are associated with a single interrupt request source, called the ADC interrupt request. Any product that includes Function \$36 allocates an ADC interrupt request bit in the Interrupt Status register (see Function \$01), and an ADC interrupt enable bit in the Interrupt Enable register (see Function \$01). The position of the ADC interrupt bits in those registers is product-specific; consult the product-specific documentation to determine their location.

12. Standard RMI physical layers

RMI is defined so that it may be implemented atop a variety of physical interfaces.

Initially, RMI is defined for three physical layers:

- RMI on I²C: See section 12.1.
- RMI on-SMBus: See section 12.2.
- RMI on four-wire SPI: See section 12.3.

12.1. I²C physical interface

Synaptics RMI-on-I²C devices are suitable for connecting directly to an industry-standard I²C host interface. This section describes the I²C physical layer. The RMI-on- I²C interface has been developed using version 2.1 of the *I²C Bus Specification*, dated January 2000. This document can be found at <http://www.nxp.com/>. The remainder of this section assumes that the reader has familiarity with the *I²C Bus Specification* document.

12.1.1. I²C transfer protocols

To communicate with an RMI-on-I²C device, a host needs to be able to:

1. Read one or more RMI registers starting from some RMI register address.
2. Write one or more RMI registers starting from some RMI register address.

The I²C bus specification imposes no limit to the number of registers that the host can read in a single transfer. However, RMI does not permit a physical layer transfer to cross a page boundary, so the practical limit on the maximum length of a transfer would be the number of registers in an RMI address page, or 256.

The I²C bus specification imposes no limit on how long a transfer can take, or how slowly a bus master is allowed to clock the bus. To meet this specification, Synaptics RMI devices will not impose any timeouts during any I²C transfer.

12.1.1.1. I²C transfer details

The following terms are used in the definition of the I²C transfer protocols:

Table 9. I²C transfer protocol terms

<i>S</i>	Indicates an I ² C “Start” Event
<i>P</i>	Indicates an I ² C “Stop” Event
<i>Sr</i>	Indicates an I ² C “Repeated Start” Event
<i>A</i>	Indicates an I ² C “ACK” bit
<i>N</i>	Indicates an I ² C “NAK” bit
<i>SlaveAddr</i>	The 7-bit Slave Address field in an I ² C header byte
<i>Wr</i>	The 1-bit ‘write’ field in an I ² C header byte (a Write always has the value 0)
<i>Rd</i>	The 1-bit ‘read’ field in an I ² C header byte (a Read always has the value 1)

12.1.2. RMI register addressing

In order to minimize bus traffic, only the low 8-bits of the full 16-bit RMI register addresses is specified by a read or write transfer. The high 8-bits of the address are supplied by the Page Select register (see section 2.3.2.1).

12.1.3. Block read operations

The Block Read operation allows a host to read one or more RMI registers starting from a specified RMI address. The device starts reading from the RMI address specified by the read operation, and continues to send registers from consecutively incrementing RMI addresses until the host finally NAKs the transfer.

The following is an example of a Block Read operation; the host reads 1 RMI register from address N:

S	SlaveAddr	Wr	A	Low 8 bits of register addr N	A	Sr	Slave Addr	Rd	A	Register N	N	P
---	-----------	----	---	-------------------------------	---	----	------------	----	---	------------	---	---

The following is an example of a Block Read operation, where the host reads 4 consecutive RMI registers starting from address N:

S	SlaveAddr	Wr	A	Low 8 bits of register addr N	A	Sr	Slave Addr	Rd	A	Register N	A	Register N+1	A
---	-----------	----	---	-------------------------------	---	----	------------	----	---	------------	---	--------------	---

Register N+2	A	Register N+3	N	P
--------------	---	--------------	---	---

It is not permitted to perform an I²C read operation that is not preceded by an I²C write of the low-order 8 bits of the RMI register address. The result of doing so is undefined.

12.1.3.1. Repeated starts

The Repeated Start separating the write of the RMI register address from the read of the register data ensures correct operation on an I²C bus that supports multiple bus masters. For a host bus that either does not require multi-master support or can't generate Repeated Start events, Synaptics RMI-on-I²C devices will permit a host to replace a Repeated Start with a Stop event followed by a Start event.

12.1.4. Block write operations

The Block Write operation allows a host to write one or more RMI registers starting at a specified RMI address. The device starts writing data to the RMI address specified by the write operation, and continues to write registers to consecutively incrementing RMI addresses as long as the host keeps sending data. An RMI device will acknowledge (ACK) every byte that it receives.

The following is an example of a Block Write operation, where the host writes data to a single RMI register at address N:

S	SlaveAddr	Wr	A	Low 8 bits of register addr N	A	Register N	A	P
---	-----------	----	---	-------------------------------	---	------------	---	---

The following is an example of a Block Write operation, where the host writes 3 consecutive RMI registers starting with the register at address N:

S	SlaveAddr	Wr	A	Low 8 bits of register addr N	A	Register N	A	Register N+1	A	Register N+2	A	P
---	-----------	----	---	-------------------------------	---	------------	---	--------------	---	--------------	---	---

12.1.5. I²C protocol compliance

The Synaptics I²C interface is designed to comply with the basic I²C protocol as described in the *I²C Bus Specification*, Version 2.1 by Philips. Conforming to this specification ensures the following:

- Synaptics modules correctly recognize and respond to Start events, Repeated Start events, and Stop events.
- Synaptics devices properly generate SCL “clock stretching” as a slave device.
- Synaptics modules support the 7-bit addressing mode.

12.1.5.1. Addressing modes

Synaptics devices do not support the 10-bit addressing extension to I²C. Synaptics Master-Slave modules master their transmissions to a host device using 7-bit addresses.

Synaptics modules can co-exist on the same I²C bus with other devices that support the 10-bit extended addressing mode. In addition, while Synaptics Slave-Only modules have 7-bit addresses, they can respond as slave device to a host/master that has a 10-bit address.

12.1.5.2. Data rate and clock stretching

Synaptics I²C devices can maintain either the “Fast Mode” data rate in the *I²C Bus Specification* at 400K bits per second, or the “Normal Mode” data rate of 100K bits per second while a byte is being clocked over the bus. In either case, a Synaptics RMI device may be required to perform an I²C Clock-Stretch operation in between the bytes of a transfer.

12.2. SMBus physical interface

Synaptics RMI-on-SMBus devices are suitable for connecting directly to an industry-standard SMBus host interface. This section describes the SMBus physical layer. The RMI-on-SMBus interface has been developed using version 2.0 of the *System Management Bus (SMBus) Specification*, dated August 3, 2000. This document can be found at <http://www.smbus.org/>.

SMBus represents a layer on top of a standard I²C interface. The main contribution of the SMBus layer is to define a set of standardized I²C transaction sequences (called SMBus ‘transfer protocols’) that are used to read or write data to a SMBus device. The SMBus transaction protocols supported by the Synaptics RMI-on-SMBus interface are defined in section 12.2.2.

12.2.1. RMI-on-SMBus addressing

RMI defines a 16-bit RMI register address space. When RMI is implemented using the SMBus physical layer, the size of the SMBus *Command Code* effectively limits the size of a register address to 8-bits. The Page Select register (see section 2.3.2.1) is used to supply the upper 8 bits of the 16-bit RMI address.

12.2.2. SMBus transfer protocols

To communicate with an RMI-on-SMBus device, a host needs to be able to do the following things:

1. Read one or more RMI registers starting from some RMI register address.
2. Write one or more RMI registers starting from some RMI register address.

RMI-on-SMBus supports the standard SMBus transfer protocols to read and write bytes, and to read and write words. Because all RMI registers are 8-bit registers, reading or writing a word really means to read or write a pair of sequential RMI byte-wide registers. The following subsections describe the SMBus read and write commands, using the notation found in the *System Management Bus (SMBus) Specification*, Version 2.0 of August 3, 2000.

12.2.2.1. SMBus transfer details

The following terms are used in the definition of the SMBus transfer protocols:

Table 10. SMBus transfer protocol terms

<i>S</i>	Indicates an SMBus “Start” Event
<i>P</i>	Indicates an SMBus “Stop” Event
<i>Sr</i>	Indicates an SMBus “Repeated Start” Event
<i>A</i>	Indicates an SMBus “ACK” bit
<i>N</i>	Indicates an SMBus “NAK” bit
<i>SlaveAddr</i>	The 7-bit Slave Address field in an SMBus header byte
<i>Wr</i>	The 1-bit ‘write’ field in an SMBus header byte (a Write always has the value 0)
<i>Rd</i>	The 1-bit ‘read’ field in an SMBus header byte (a Read always has the value 1)

12.2.2.2. SMBus Byte Write

The SMBus *Byte Write* command writes a byte to a single register in an RMI-on-SMBus device. In its general form, the SMBus *Byte Write* command has this format:

S	SlaveAddr	Wr	A	Command Code (register addr)	A	Data Byte	A	P
---	-----------	----	---	---------------------------------	---	-----------	---	---

- The *Command Code* contains the low 8 bits of the address of the RMI register to be written. The Page Select register supplies the high 8 bits of the address. The result is a 16-bit RMI address ‘N’.
- The *Data Byte* is the value to write to the RMI register at address ‘N’.

12.2.2.3. SMBus Byte Read

The SMBus *Byte Read* command reads the contents of single register in an RMI-on-SMBus device. In its general form, the SMBus *Byte Read* command has this format:

S	SlaveAddr	Wr	A	Command Code (register addr)	A	Sr	Slave Addr	Rd	A	Data Byte	N	P
---	-----------	----	---	---------------------------------	---	----	------------	----	---	-----------	---	---

- The *Command Code* contains the low 8 bits of the address of the RMI register to be written. The Page Select register supplies the high 8 bits of the address. The result is a 16-bit RMI address ‘N’.
- The *Data Byte* is the value that was read from the RMI register at address ‘N’.
- All SMBus Read operations terminate with a NAK bit followed by a STOP bit.

12.2.2.4. SMBus Word Write

The SMBus *Word Write* command writes a pair of bytes to a sequential pair of registers in an RMI-on-SMBus device. In its general form, the SMBus *Word Write* command has this format:

S	SlaveAddr	Wr	A	Command Code (register addr)	A	Data Byte 0	A	Data Byte 1	A	P
---	-----------	----	---	---------------------------------	---	-------------	---	-------------	---	---

- The *Command Code* contains the low 8 bits of the address of the first RMI register to be written. The Page Select register supplies the high 8 bits of the address. The result is a 16-bit RMI address ‘N’.
- *Data Byte 0* is written to the RMI address ‘N’.
- *Data Byte 1* is written to the RMI address ‘N’+1.

12.2.2.5. SMBus Word Read

The SMBus *Word Read* command reads the contents of a sequential pair of registers in an RMI-on-SMBus device. In its general form, the SMBus *Word Read* command has this format:

S	SlaveAddr	Wr	A	Command Code (register addr)	A	Sr	Slave Addr	Rd	A	Data Byte 0	A	Data Byte 1	N	P
---	-----------	----	---	---------------------------------	---	----	------------	----	---	-------------	---	-------------	---	---

- The *Command Code* contains the low 8 bits of the address of the first RMI register to be written. The Page Select register supplies the high 8 bits of the address. The result is a 16-bit RMI address ‘N’.

- *Data Byte 0* is the contents of the register at the RMI address ‘N’.
- *Data Byte 1* is the contents of the register at the RMI address ‘N’+1.
- All SMBus Read operations terminate with a NAK bit followed by a STOP bit.

12.2.3. Multi-register block read/write operations

To improve bus utilization, Synaptics RMI-on-SMBus devices permit special multi-register read and write operations as an extension to the SMBus specification. If a host performs a standard Byte Read operation but simply keeps reading more bytes, the device will continue to send registers from consecutively incrementing RMI addresses until the host finally NAKs the transfer.

Below is an example of a multi-register Read operation, where the host reads 4 consecutive RMI registers starting from the specified register address ‘N’:

S	SlaveAddr	Wr	A	Command Code (register addr)	A	Sr	Slave Addr	Rd	A	Register N	A	Register N+1	A
---	-----------	----	---	---------------------------------	---	----	------------	----	---	------------	---	--------------	---

Register N+2	A	Register N+3	N	P
--------------	---	--------------	---	---

In similar fashion, if the host performs a standard Byte Write operation but simply keeps writing more bytes, the RMI-on-SMBus device will write the extra bytes to subsequent register addresses.

Below is an example of a multi-register Write operation, where the host writes 3 consecutive RMI registers starting with register address N:

S	SlaveAddr	Wr	A	Command Code (register addr)	A	Register N	A	Register N+1	A	Register N+2	A	P
---	-----------	----	---	---------------------------------	---	------------	---	--------------	---	--------------	---	---

12.2.4. Repeated starts

For the Read Byte and Read Word transfer protocols, the SMBus specification requires that a Repeated Start event must be used to separate the writing of the Command Code byte from the reading of the data byte(s). The Repeated Start ensures correct operation in host systems that support multiple bus masters. For host systems that either do not require multi-master support or can’t generate Repeated Start events, Synaptics RMI-on-SMBus devices permit a host to replace a Repeated Start with a Stop event followed by a Start event.

12.2.5. SMBus compliance

The SMBus Specification Version 2.0 describes a wide variety of features. Not all of these features are required to be supported for a particular device to be considered to be SMBus-compliant. Full information on the SMBus can be found in the document titled *System Management Bus (SMBus) Specification*, Version 2.0 of August 3, 2000. This document can be found at <http://www.smbus.org/>.

The SMBus is described in terms of *layers*. Synaptics SMBus compliance issues are dealt with on a layer-by-layer basis.

12.2.5.1. Layer 1: physical layer

In general, the SMBus physical layer looks like a standard I²C physical layer interface. To solve certain problems inherent in a typical I²C interface, SMBus imposes some additional restrictions on the I²C interface that it uses as its physical layer. These restrictions typically have to do with the timing and length of the transfers that are permitted. The important restrictions imposed by the SMBus specification on a Synaptics RMI slave device are:

- 10 KHz minimum bus operating frequency,
- 25 mSec (min), 35 mSec (max) clock-low timeout period, and
- 500 mSec (max) Time in which a device must be operational after power-on reset.

Note: Synaptics RMI-on-SMBus devices do not support the SMBus SMBALERT# signal. This means that Synaptics SMBus devices will not respond to the SMBus Alert Response Address. Synaptics devices support a general purpose Attention signal (ATTN) that can either be used as an interrupt input to a host processor or as an input that the host can poll. As an order-time option, the ATTN signal can be configured as being either active high or active low.

12.2.5.2. Layer 2: data link layer

Synaptics RMI-on-SMBus devices implement the Data Link layer as described in the *SMBus Specification*. Section 4.3.3 of the *SMBus Specification* Version 2.0 defines that SMBus devices must implement “clock-low extending” (also known as I²C “clock-stretching”). In particular, the *SMBus Specification* V2.0 defines that *all* devices on a SMBus (both masters and slaves) must be able to tolerate both periodic and random clock stretching.

Synaptics RMI-on-SMBus slave devices can tolerate both random and periodic clock-stretching imposed by any other device sharing the SMBus.

Synaptics SMBus slave devices stretch the clock for short periods of time in random fashion, but they never stretch the clock long enough to violate the 10 KHz (min) bus transfer frequency.

12.2.5.3. Layer 3: SMBus network layer

The SMBus Specification defines eleven different command protocols that can be used to transfer data. The specification states that a slave device does not need to support all eleven protocols in order to be SMBus compliant.

Synaptics RMI-on-SMBus devices support the following four SMBus transfer protocols:

- Byte Read, Byte Write
- Word Read, Word Write

For increased transfer efficiency, Synaptics RMI-on-SMBus devices extend the SMBus protocol by supporting special multi-register reads and writes (see section 12.2.3).

Note: Synaptics SMBus devices do not support the ARP functionality to assign bus addresses. Synaptics SMBus devices implement a fixed, 7-bit I²C addressing mechanism. The 7-bit I²C slave address for a given Synaptics SMBus device is fixed at the time that the product is ordered. It is the implementer's responsibility to choose a SMBus address that will not conflict with other SMBus devices in their system. If desired, RMI-on-SMBus modules can be ordered with an address-strapping option. This allows a host system to select a module's I²C address among two or more preconfigured I²C addresses by strapping module IO pins.

Also, Synaptics devices do not support the SMBus *Packet Error Check (PEC)* byte. Hosts should not expect Synaptics devices to generate a *PEC* byte during read operations. Hosts should not send *PEC* bytes to a Synaptics device during write operations.

12.3. SPI physical interface

This section describes the RMI-on-SPI physical layer.

12.3.1. SPI signals

RMI on SPI uses the industry-standard four-wire SPI interface. The SPI signals include:

- SSB, a device-select signal driven by the host. In some SPI systems this signal is known as Slave Select, **SS**, or Chip Select, **CS**. SSB is an active-low signal that goes low when an RMI transaction is in progress.
- SCK, a clock signal driven by the host. Several clocking conventions are supported, as described in section 12.3.2.
- MOSI (master out / slave in), a data signal driven by the host.
- MISO (master in / slave out), a data signal driven by the RMI device. The device drives MISO only when SSB is low; when SSB is high, the device floats its MISO pin. This allows multiple RMI devices to be connected with SCK, MOSI, and MISO all tied in parallel, using separate SSB wires to address the various devices.
- ATTN, an *optional* attention signal driven by the RMI device. This pin is not present on devices that use the SRQ mechanism to signal attention (see section 12.3.4).
- RESET, an *optional* reset signal driven by the host. If a device provides a RESET pin, the pin is an active-low input with a pull-up resistor on the RMI device. This allows RESET to be left unconnected when not needed.

12.3.2. SPI clocking

“SPI” is actually a loosely defined family of standard interfaces. The clock polarity and clock phase (often denoted CPOL and CPHA) vary from one SPI system to another. Synaptics can supply RMI devices that use the standard clocking conventions that correspond to CPHA = ‘1’. The clocking conventions that correspond to CPHA = ‘0’ are not currently supported by the RMI standard.

CPOL defines the idle level of SCK. CPOL is ‘0’ if SCK is low between transactions, or ‘1’ if SCK is high between transactions. The usual clocking convention for RMI-on-SPI devices is CPOL = ‘1’; upon request, Synaptics can also supply RMI devices that use the CPOL = ‘0’ clocking convention.

CPHA defines on which SCK edge the MOSI data and MISO data are sampled by their respective receivers.

CPHA is '0' to sample on the edge where SCK leaves its idle level (for example., the falling edge if CPOL is '1'), or CPHA is '1' to sample on the edge where SCK returns to its idle level (the rising edge if CPOL is '1'). Thus, all current RMI-on-SPI devices sample MOSI, and expect the host to sample MISO, on the trailing SCK edge where SCK returns to its idle level.

RMI always transmits each byte most-significant-bit first, following the convention of most chips' SPI interfaces. RMI always changes both MISO and MOSI on the same clock edge, and it always samples both MISO and MOSI on the same (opposite) clock edge.

12.3.3. SPI transaction format

The host (the SPI master) drives the SSB pin high between transactions, and low during a transaction (see Figure 78 and Figure 79). When SSB is high, the device (the SPI slave) floats its MISO pin and ignores the MOSI and SCK pins; this allows multiple devices to share the MISO, MOSI, and SCK pins provided that each device receives a separate SSB signal. When SSB is low, the device drives the MISO pin, and it samples MOSI and changes MISO in response to the clock waveform on SCK. SSB edges delimit a transaction. The host is not allowed to tie SSB low permanently: it must fall to begin a transaction, and rise to complete the transaction.

During the first two bytes (16 SCK pulses) after the fall of SSB, the host transmits an *address word* on MOSI, most significant byte first. In the address word, bit 15 is '1' for a read transaction and '0' for a write transaction. Bits 14:0 hold the register address *R*. During the first two bytes of the address word, the device transmits undefined data on MISO.

The SPI physical layer is only capable of transferring a 15-bit address. If an RMI device is constructed that requires access to the full 16-bit address space, the addresses above \$7FFF will have to be accessed via setting the Page Select register (see section 2.3.2.1).

For a read transaction, in subsequent bytes (groups of 8 SCK pulses), the device transmits on MISO the contents of consecutive registers starting from the addressed register, and MOSI is ignored. It is permissible for SSB to be driven high between the second and third bytes of a read transaction (between the "address write" and "data read" portions of the transaction), but it is recommended to hold SSB low for the entire transaction if possible.

For a write transaction, in subsequent bytes, the host transmits on MOSI the write data for consecutive registers starting from the addressed register, and the device transmits undefined data on MISO. During a write transaction that writes several consecutive registers, the writing action to each register occurs as the transfer of the data byte for the register completes (except for a very few multi-byte quantities that are written only when the final byte is written, as described in section 2.1).

Note: This addressing mechanism is different from that of RMI-on-SMBus, but it is more consistent with the types of mechanisms most often used on SPI devices.

The transaction ends when the host raises SSB. If the host raises SSB during or after either byte of the address word, no transaction occurs. If the host raises SSB during a write transaction when only a fraction of the 8 bits of a data byte have been transmitted, the register corresponding to that data byte is not written (but any registers written earlier in the transaction will already be committed).

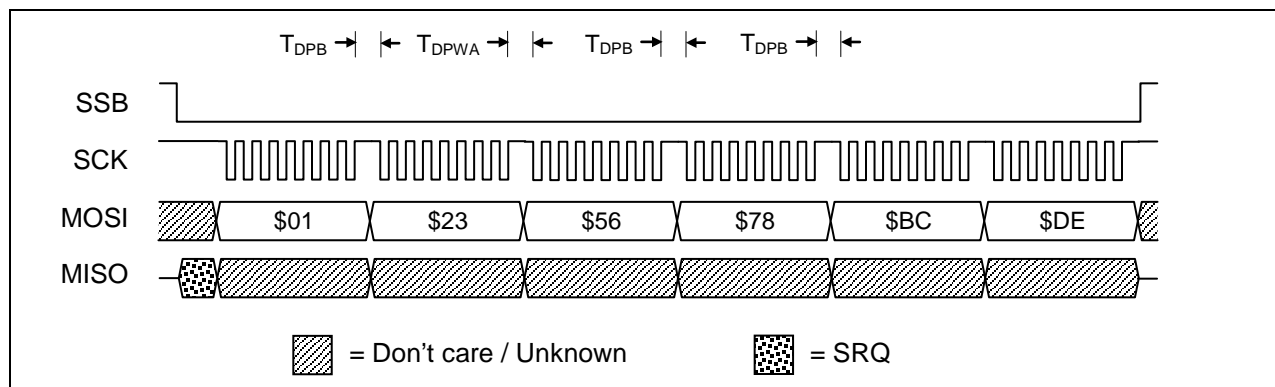


Figure 78. RMI-on-SPI write transaction (assuming $CPOL = '1'$ and $CPHA = '1'$)

Note: The Host writes \$56, \$78, \$BC, and \$DE to registers \$0123–\$0126, respectively.

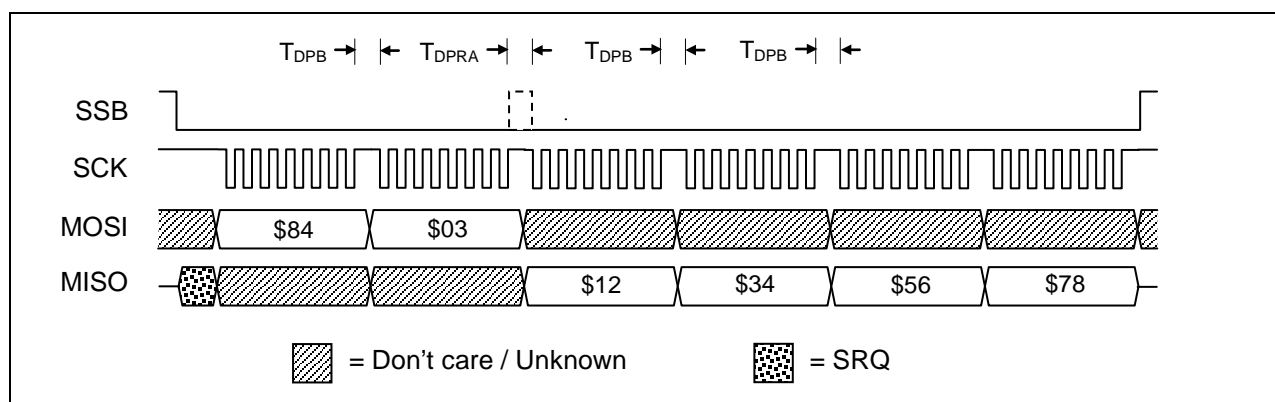


Figure 79. RMI-on-SPI read transaction (assuming $CPOL = '1'$ and $CPHA = '1'$)

Notes:

- The Host reads \$12, \$34, \$56, and \$78 from registers \$0403–\$0406, respectively.
- It is recommended that SSB remain low for the entire transaction, but it is permissible for SSB to drive high between the “address write” and “data read” portions of the transaction.
- See the product datasheet (for example, the ClearPad 3000 Series Platform Datasheet) for the minimum duration for T_{DPWA} , T_{DPRA} , and T_{DPB} .

12.3.4. SPI attention mechanism

Synaptics can offer RMI devices using either of two additional attention mechanisms. One mechanism uses a non-standard extension of the SPI interface called a “service request” (SRQ) bit. The other uses a separate ATTN pin to hold the attention signal.

Note: The option for a separate attention pin accommodates hosts that can’t handle interrupts and MISO signals on the same host pin. It also may help in case RMI device interfaces must be implemented in non-Synaptics chips that can’t generate an SRQ-like bit on MISO.

In the SRQ option, the RMI device drives the attention signal onto MISO as soon as SSB falls (see Figure 78 and Figure 79).

The SRQ attention signal is “live” on MISO in the interval between the fall of SSB and the first SCK edge: The host may lower SSB, leave SCK at its idle level, and watch MISO using an interrupt to wait for an interrupt request report from the device. After the first SCK edge, it is undefined whether MISO continues to follow the “live” attention state, or freezes at the state of the attention signal at the time of the SCK edge.

Note: The SRQ signal may be “live,” but its behavior is relatively simple: The only change to MISO that can possibly occur during the SRQ period is one transition from the inactive to the active attention level, because attention, once asserted, can be deasserted only by a host transaction that reads the data registers or writes the Interrupt Enable bits. For RMI devices implemented using Synaptics’ SPI-compatible chips, MISO will freeze after the first SCK edge.

Synaptics is also able to supply RMI devices that transmit the attention signal on a fifth ATTN pin. In this option, ATTN can be ordered either as an active-high push-pull output pin, or as an active-low open-drain pin for which the host must supply an external pull-up resistor. (The latter option allows multiple RMI devices’ ATTN pins to be merged in a wired-OR configuration. Because MISO is a fully driven push-pull output, the ATTN attention mechanism is better suited than SRQ to multi-device RMI systems.)

Contact Us

To locate the Synaptics office nearest you, visit our website at www.synaptics.com.

